DevOps

Na prática: entrega de software confiável e automatizada

DANILO SATO

"Para meu pai, que me introduziu ao mundo da computação e é meu exemplo de vida."

Prefácio

Jez Humble

Pouco depois que me formei na universidade em 1999, eu fui contratado por uma *start-up* em Londres. Meu chefe, Jonny LeRoy, ensinou-me a prática de implantação contínua: quando terminávamos uma nova funcionalidade, fazíamos alguns testes manuais rápidos na nossa estação de trabalho e, em seguida, copiávamos os *scripts* ASP relevantes por FTP para o servidor de produção – uma prática que eu não recomendaria hoje, mas que teve a vantagem de nos permitir mostrar novas ideias para nossos usuários muito rapidamente.

Em 2004, quando entrei na ThoughtWorks, meu trabalho era ajudar empresas a entregar software e eu fiquei chocado ao descobrir que prazos de meses ou mesmo anos eram comuns. Felizmente, tive a sorte de trabalhar com várias pessoas inteligentes em nossa indústria que estavam explorando formas de melhorar estes resultados, ao mesmo tempo aumentando a qualidade e melhorando a nossa capacidade de servir nossos usuários. As práticas que desenvolvemos também tornaram a vida melhor para as pessoas com quem estávamos trabalhando (por exemplo, não precisávamos mais fazer *deploys* fora do horário comercial) – uma indicação importante de que você está fazendo algo certo. Em 2010, Dave Farley e eu publicamos *Entrega Contínua*, onde descrevemos os princípios e práticas que tornam possível entregar pequenas alterações incrementais, de forma rápida, barata e com baixo risco.

No entanto, o nosso livro omite os detalhes práticos do que você realmente precisa para começar a criar uma pipeline de entrega, como pôr em prática sistemas de monitoramento e infraestrutura como código, além dos outros passos práticos importantes necessários para implementar entrega contínua. Por isso estou muito contente que o Danilo escreveu o livro que está

em suas mãos, que eu acho ser uma contribuição importante e valiosa para a nossa área. O Danilo está profundamente envolvido em ajudar organizações a implementar as práticas de entrega contínua há vários anos e tem ampla experiência, e eu tenho certeza de que você vai achar o seu livro prático e informativo. Desejo-lhe tudo de melhor na sua jornada.

Sobre o livro

Entregar software em produção é um processo que tem se tornado cada vez mais difícil no departamento de TI de diversas empresas. Ciclos longos de teste e divisões entre as equipes de desenvolvimento e de operações são alguns dos fatores que contribuem para este problema. Mesmo equipes ágeis que produzem software entregável ao final de cada iteração sofrem para chegar em produção quando encontram estas barreiras.

DevOps é um movimento cultural e profissional que está tentando quebrar essas barreiras. Com o foco em automação, colaboração, compartilhamento de ferramentas e de conhecimento, DevOps está mostrando que desenvolvedores e engenheiros de sistema têm muito o que aprender uns com os outros.

Neste livro, mostramos como implementar práticas de DevOps e Entrega Contínua para aumentar a frequência de deploys na sua empresa, ao mesmo tempo aumentando a estabilidade e robustez do sistema em produção. Você vai aprender como automatizar o build e deploy de uma aplicação web, como automatizar o gerenciamento da infraestrutura, como monitorar o sistema em produção, como evoluir a arquitetura e migrá-la para a nuvem, além de conhecer diversas ferramentas que você pode aplicar no seu trabalho.

Agradecimentos

Ao meu pai, Marcos, por ser sempre um exemplo a seguir e por ir além tentando acompanhar os exemplos de código mesmo sem nenhum conhecimento no assunto. À minha mãe, Solange, e minha irmã, Carolina, pelo incentivo e por corrigirem diversos erros de digitação e português nas versões preliminares do livro.

À minha parceira e melhor amiga, Jenny, pelo carinho e apoio durante as diversas horas que passei trabalhando no livro.

Ao meu editor, Paulo Silveira, pela oportunidade, pela confiança e por saber como dar um puxão de orelha ou um incentivo na hora certa para que o livro se tornasse uma realidade. À minha revisora e amiga, Vivian Matsui, por corrigir todos os meus erros de português.

Aos meus revisores técnicos: Hugo Corbucci, Daniel Cordeiro e Carlos Vilella. Obrigado por me ajudarem a encontrar formas melhores de explicar conceitos difíceis, pela opinião sobre os termos difíceis de traduzir, por questionarem minhas decisões técnicas e por me ajudarem a melhorar o conteúdo do livro.

Aos colegas Prasanna Pendse, Emily Rosengren, Eldon Almeida e outros membros do grupo "Blogger's Bloc" na ThoughtWorks, por me incentivarem a escrever mais e pelo *feedback* nos capítulos iniciais, mesmo não entendendo a língua portuguesa.

Aos meus inúmeros outros colegas de trabalho da ThoughtWorks, em especial Rolf Russell, Brandon Byars e Jez Humble, que ouviram minhas ideias sobre o livro e me ajudaram a escolher a melhor forma de abordar cada assunto, capítulo por capítulo.

Por fim, a todos que contribuíram de forma direta ou indireta na escrita deste livro.

Muito obrigado!

Sobre o autor

Danilo Sato começou a programar ainda criança, quando muitas pessoas ainda não tinham computador em casa. Em 2000 entrou no curso de bacharelado em Ciência da Computação da Universidade de São Paulo, começando sua carreira como administrador da Rede Linux do IME-USP durante 2 anos. Ainda na graduação começou a estagiar como desenvolvedor Java/J2EE e teve seu primeiro contato com Métodos Ágeis na disciplina de Programação Extrema (XP).

Iniciou o mestrado na USP logo após a graduação e, orientado pelo Professor Alfredo Goldman, defendeu em Agosto de 2007 a dissertação "Uso Eficaz de Métricas em Métodos Ágeis de Desenvolvimento de Software" [16].

Durante sua carreira, Danilo atuou como consultor, desenvolvedor, administrador de sistemas, analista, engenheiro de sistemas, professor, arquiteto e *coach*, tornando-se consultor líder da ThoughtWorks em 2008, onde trabalhou em projetos Ruby, Python e Java no Brasil, EUA e no Reino Unido. Atualmente tem ajudado clientes a adotar práticas de DevOps e Entrega Contínua para reduzir o tempo entre a concepção de uma ideia e sua implementação com código rodando em produção.

Além da carreira profissional, Danilo também tem experiência como palestrante em conferências nacionais e internacionais, apresentando palestras e *workshops* na: XP 2007/2009/2010, Agile 2008/2009, Ágiles 2008, Conexão Java 2007, Falando em Agile 2008, Rio On Rails 2007, PyCon Brasil 2007, RejectConf SP 2007, Rails Summit Latin America 2008, Agile Brazil 2011/2012/2013, QCon SP 2011/2013, RubyConf Brasil 2012/2013, além de ser o fundador do Coding Dojo @ São Paulo e organizador da Agile Brazil 2010, 2011 e 2012.

Sumário

Capítulo 1

Introdução

Com o avanço da tecnologia, software tem se tornado parte fundamental do dia a dia de muitas empresas. Ao planejar suas férias em família – agendando quarto em hotéis, comprando passagens de avião, realizando transações financeiras, enviando SMS ou compartilhando as fotos da viagem – você interage com diversos sistemas de software. Quando um desses sistemas está fora do ar, o problema não é apenas da empresa que está perdendo negócios, mas também dos usuários que não conseguem realizar suas tarefas. Por esse motivo é importante investir na qualidade e estabilidade do software desde o momento em que a primeira linha de código é escrita até o momento em que ele está rodando em produção.

1.1 Abordagem tradicional

Metodologias de desenvolvimento de software evoluíram, porém o processo de transformar ideias em código ainda envolve diversas atividades como: levantamento de requisitos, *design*, arquitetura, implementação e teste. Os Métodos Ágeis de desenvolvimento de software surgiram no final da década de 90 propondo uma nova abordagem para organizar tais atividades. Ao invés de realizá-las em fases distintas – modelo conhecido como processo em cascata – elas acontecem em paralelo o tempo todo, em iterações curtas. Ao final de cada iteração, o software se torna mais e mais útil, com novas funcionalidades e menos *bugs*, e o time decide junto com o cliente qual a próxima fatia a ser desenvolvida.

Assim que o cliente decide que o software está pronto para ir ao ar e o código é colocado em produção, os verdadeiros usuários começam a usar o sistema. Nesse momento, diversas outras preocupações se tornam relevantes: suporte, monitoramento, segurança, disponibilidade, desempenho, usabilidade, dentre outras. Quando o software está em produção, a prioridade é mantê-lo rodando de forma estável. Em casos de falha ou desastres, o time precisa estar preparado para reagir de forma proativa e resolver o problema rapidamente.

Devido à natureza dessas atividades, muitos departamentos de TI possuem uma divisão clara de responsabilidades entre o **time de desenvolvimento** e o **time de operações**. Enquanto o time de desenvolvimento é responsável por criar novos produtos e aplicações, adicionar funcionalidades ou corrigir *bugs*, o time de operações é responsável por cuidar desses produtos e aplicações em produção. O time de desenvolvimento é incentivado a introduzir mudanças, enquanto o time de operações preza pela estabilidade.

À primeira vista, esta divisão de responsabilidades parece fazer sentido. Cada equipe possui objetivos e formas de trabalho diferentes. Enquanto o time de desenvolvimento trabalha em iterações, o time de operações precisa reagir instantaneamente quando algo dá errado. Além disso, as ferramentas e o conhecimento necessário para trabalhar nessas equipes são diferentes. O time de desenvolvimento evolui o sistema introduzindo mudanças. Por outro lado, o time de operações evita mudanças, pois elas trazem um certo risco à estabilidade do sistema. Cria-se então um conflito de interesse entre essas

duas equipes.

Uma vez que o conflito existe, a forma mais comum de gerenciar essa relação é através da criação de processos que definem o modo de trabalho e as responsabilidades de cada equipe. De tempos em tempos, a equipe de desenvolvimento empacota o software que precisa ir para produção, escreve documentação explicando como configurar o sistema, como realizar a instalação em produção e repassa a responsabilidade para o time de operações. É comum o uso de sistemas de controle de *tickets* para gerenciar a comunicação entre as equipes assim como a definição de acordos de nível de serviço (*service-level agreements* ou SLAs) para garantir que os *tickets* sejam processados e respondidos em um tempo adequado.

Essa passagem de bastão – também conhecida em inglês pelo termo *hand-off* – cria um gargalo no processo de levar código do desenvolvimento e teste para produção. É comum chamar esse processo de ***deploy*** em ambiente de produção. Em português é até possível ouvir o neologismo *deploiar*, ou então o termo correto em português, que seria "implantar".

Com o passar do tempo, o processo tende a se tornar mais e mais burocrático, fazendo com que a frequência de *deploys* diminua. Com isso, o número de mudanças introduzidas em cada *deploy* tende a acumular, aumentando também o risco de cada *deploy* e criando o ciclo vicioso mostrado na figura 1.1.
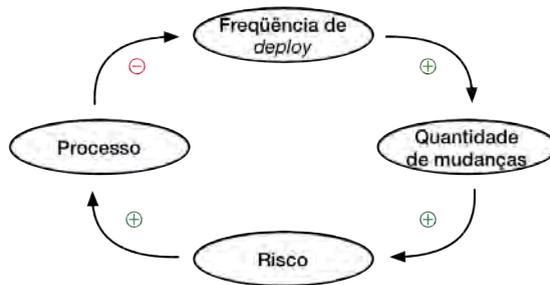
Figura 1.1: Ciclo vicioso entre desenvolvimento e operações

Esse ciclo vicioso não só diminui a habilidade da empresa de responder rapidamente a mudanças no negócio, como também impacta etapas anterio-

res do processo de desenvolvimento. A separação entre os times de desenvolvimento e de operações, o *hand-off* de código existente entre eles e a cerimônia envolvida no processo de *deploy* acabam criando o problema conhecido como "**a Última Milha**" [17].

A última milha se refere à fase final do processo de desenvolvimento que acontece após o software atender todos os requisitos funcionais mas antes de ser implantado em produção. Ela envolve várias atividades para verificar se o que vai ser entregue é estável ou não, como: testes de integração, testes de sistema, testes de desempenho, testes de segurança, homologação com usuários (também conhecido como *User Acceptance Tests* ou *UAT*), testes de usabilidade, ensaios de *deploy*, migração de dados etc.

É fácil ignorar a última milha quando a equipe está produzindo e demonstrando funcionalidades novas a cada uma ou duas semanas. Porém, são poucas as equipes que de fato fazem *deploy* em produção ao final de cada iteração. Do ponto de vista do negócio, a empresa só terá retorno no investimento quando o software estiver realmente rodando em produção. O problema da última milha só é visível quando se toma uma visão holística do processo. Para resolvê–lo é preciso olhar além das barreiras impostas entre as diferentes equipes envolvidas: seja a equipe de negócios, a equipe de desenvolvimento ou a equipe de operações.

1.2 Uma abordagem alternativa: DevOps e entrega contínua

Muitas empresas de sucesso na internet – como Google, Amazon, Netflix, Flickr, GitHub e Facebook – perceberam que a tecnologia pode ser usada a seu favor e que o atraso no *deploy* para produção significa atrasar sua habilidade de competir e se adaptar a mudanças no mercado. É comum que elas realizem dezenas ou até centenas de *deploys* por dia!

Essa linha de pensamento que tenta diminuir o tempo entre a criação de uma ideia e sua implementação em produção é também conhecida como "**Entrega Contínua**" [6] e está revolucionando o processo de desenvolvimento e entrega de software. Alguns autores preferem utilizar o termo em inglês, *continuous delivery*.

Quando o processo de *deploy* deixa de ser uma cerimônia e passa a se tornar algo corriqueiro, o ciclo vicioso da figura 1.1 se inverte completamente. O aumento da frequência de *deploys* faz com que a quantidade de mudanças em cada *deploy* diminua, reduzindo também o risco associado com aquele *deploy*. Esse benefício não é algo intuitivo, porém quando algo dá errado é muito mais fácil descobrir o que aconteceu, pois a quantidade de fatores que podem ter causado o problema é menor.

No entanto, a diminuição do risco não implica na remoção completa dos processos criados entre as equipes de desenvolvimento e de operações. O segredo que permite a inversão do ciclo é a automatização desse processo, conforme mostra a figura 1.2. Automatizar o processo de *deploy* permite que ele seja executado a qualquer momento de forma confiável, removendo o risco de um erro humano causar problemas.
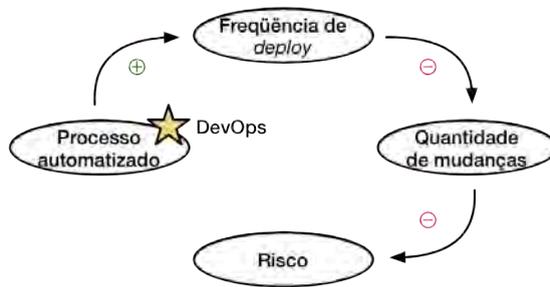
Figura 1.2: Práticas de DevOps ajudam a quebrar o ciclo vicioso através da automação de processos

Investir em automatização não é uma ideia nova, muitas equipes já escrevem testes automatizados como parte do processo de desenvolvimento de software. Práticas como o desenvolvimento dirigido a testes (*Test-Driven Development* ou TDD) [3] ou a integração contínua [12] – que será discutida em mais detalhes no capítulo 6 – são comuns e amplamente aceitas na comunidade de desenvolvimento. Esse foco em automação de testes, juntamente com a criação de equipes multidisciplinares, ajudou a quebrar a barreira existente entre desenvolvedores, testadores e analistas de negócio, criando uma cultura de colaboração entre pessoas com habilidades complementares trabalhando

como parte da mesma equipe.

Inspirado no sucesso dos métodos ágeis, um novo movimento surgiu para levar a mesma linha de raciocínio para o próximo nível: o movimento **DevOps**. Seu objetivo é criar uma cultura de colaboração entre as equipes de desenvolvimento e de operações que permite aumentar o fluxo de trabalho completado – maior frequência de *deploys* – ao mesmo tempo aumentando a estabilidade e robustez do ambiente de produção.

Além de uma mudança cultural, o movimento DevOps enfoca bastante nas práticas de automação das diversas atividades necessárias para atacar a última milha e entregar código de qualidade em produção, como: compilação do código, testes automatizados, empacotamento, criação de ambientes para teste ou produção, configuração da infraestrutura, migração de dados, monitoramento, agregamento de *logs* e métricas, auditoria, segurança, desempenho, *deploy*, entre outros.

Empresas que aplicaram essas práticas de DevOps com sucesso não enxergam mais o departamento de TI como um gargalo mas sim como um agente de capacitação do negócio. Elas conseguem se adaptar a mudanças no mercado rapidamente e realizar diversos *deploys* por dia de forma segura. Algumas delas inclusive fazem com que um novo desenvolvedor realize um *deploy* para produção no seu primeiro dia de trabalho!

Este livro irá apresentar, através de exemplos reais, as principais práticas de DevOps e Entrega Contínua para permitir que você replique o mesmo sucesso na sua empresa. O objetivo principal do livro é aproximar as comunidades de desenvolvimento e de operações. Desenvolvedores conhecerão um pouco mais sobre as preocupações e práticas envolvidas para operar e manter sistemas estáveis em produção, enquanto engenheiros e administradores de sistema irão aprender como introduzir mudanças de forma segura e incremental através de automação.

1.3 Sobre o livro

O principal objetivo do livro é mostrar na prática como aplicar os conceitos e técnicas de DevOps e Entrega Contínua. Por esse motivo, tivemos que escolher quais tecnologias e ferramentas utilizar. Nossa preferência foi usar

linguagens e ferramentas de código livre e priorizamos aquelas que são bastante usadas na indústria.

Você vai precisar usar as ferramentas escolhidas para acompanhar os exemplos de código. Porém, sempre que apresentarmos uma nova ferramenta, vamos discutir um pouco sobre as outras alternativas, para que você possa pesquisar qual opção faz mais sentido no seu contexto.

Você não precisa de nenhum conhecimento prévio específico para acompanhar os exemplos. Se você tiver experiência no ecossistema Java ou Ruby, será um bônus. Nosso ambiente de produção vai rodar em UNIX (Linux, para ser mais específico), portanto um pouco de conhecimento no uso da linha de comando pode ajudar mas não é obrigatório [13]. De toda forma, você vai ser capaz de rodar todos os exemplos na sua própria máquina, independente se está rodando Linux, Mac ou Windows.

Público alvo

Este livro é voltado para desenvolvedores, engenheiros de sistema, administradores de sistema, arquitetos, gerentes e qualquer pessoa com conhecimentos técnicos que tenha interesse em aprender mais sobre as práticas de DevOps e Entrega Contínua.

Estrutura dos capítulos

O livro foi escrito para ser lido do início ao fim de forma sequencial. Dependendo da sua experiência com o assunto abordado em cada capítulo, você pode preferir avançar ou seguir em uma ordem diferente.

No **capítulo 2** apresentamos a aplicação de exemplo que será usada no restante do livro e as tecnologias usadas para construí-la. Como o foco do livro não é no desenvolvimento da aplicação em si, usamos uma aplicação não trivial escrita em Java e usando diversas bibliotecas bem comuns no ecossistema Java. Ao final do capítulo, a aplicação vai estar rodando em produção.

Com um ambiente de produção no ar, no **capítulo 3** vamos vestir o chapéu do time de operações e configurar um servidor de monitoramento para detectar falhas e enviar notificações sempre que algum problema for encontrado. Ao final do capítulo, seremos notificados de que um dos servidores caiu.

No **capítulo 4** vamos reconstruir o servidor problemático, dessa vez de forma automatizada, tratando a infraestrutura como código. O **capítulo 5** é uma continuação do assunto, abordando tópicos mais avançados e refatorando o código para torná-lo mais modular, legível e extensível.

Após vestir o chapéu do time de operações, iremos voltar nossa atenção para o desenvolvimento de software. O **capítulo 6** discute práticas de engenharia ágil que ajudam a escrever código de qualidade. Vamos aprender sobre os diversos tipos de testes automatizados e subir um novo servidor dedicado para fazer integração contínua do código da nossa aplicação.

Apresentamos o conceito da pipeline de entrega no **capítulo 7**. Colocamos o código de infraestrutura na integração contínua e implementamos um processo automatizado para instalar a versão mais nova da aplicação em produção com o simples clique de um botão.

No **capítulo 8** migramos o ambiente de produção para a nuvem. Por fim, discutimos tópicos mais avançados, com ponteiros para recursos onde você pode pesquisar e aprender mais sobre os assuntos que ficaram de fora do escopo deste livro.

Convenções de código

Nos exemplos de código, vamos usar reticências `...` para omitir as partes não importantes. Quando o código já existe, vamos repetir as linhas ao redor da área que precisa ser alterada para dar um pouco de contexto sobre a modificação.

Quando a linha é muito comprida e não couber na página, vamos usar uma barra invertida `\` para indicar que a próxima linha no livro é a continuação da linha anterior. Você pode simplesmente ignorar a barra invertida e continuar digitando na mesma linha.

Nos exemplos na linha de comando, além da barra invertida, usaremos um sinal de maior `>` na próxima linha para indicar que ela ainda é parte do mesmo comando. Isto é para distinguir o comando que precisa ser digitado da saída que o comando produz quando for executado. Ao executar os comandos, você pode simplesmente digitar tudo na mesma linha, ignorando a `\` e o `>`.

Mais recursos

Criamos um fórum de discussão no Google Groups onde você pode enviar perguntas, sugestões ou *feedback* direto para o autor. Para se inscrever, acesse a URL:

https://groups.google.com/d/forum/livro-devops-na-pratica

Todos os exemplos de código do livro também estão disponíveis no GitHub do autor, nas URLs:

https://github.com/dtsato/loja-virtual-devops

https://github.com/dtsato/loja-virtual-devops-puppet

Capítulo 2

Tudo começa em produção

Ao contrário do que muitos processos de desenvolvimento sugerem, o ciclo de vida do software só deveria começar quando usuários começam a usá-lo. O problema da última milha apresentado no capítulo 1 deveria ser a primeira milha, pois nenhum software entrega valor antes de entrar em produção. Por isso, o objetivo deste capítulo é subir um ambiente de produção completo, começando do zero, e instalar uma aplicação Java relativamente complexa que será usada como ponto de partida para os conceitos e práticas de DevOps apresentados no livro.

Ao final deste capítulo, teremos uma loja virtual web completa rodando – suportada por um banco de dados – que permitirá que usuários se cadastrem, efetuem compras, além de fornecer ferramentas de administração para gerenciamento do catálogo de produtos, das promoções e do conteúdo disponibilizado na loja virtual.

De que adianta adicionar novas funcionalidades, melhorar o desempe-

nho, corrigir defeitos, deixar as telas da loja virtual mais bonitas se isso tudo não for para produção? Vamos começar fazendo exatamente essa parte difícil, essa tal de última milha: colocar o software em produção.

2.1 Nossa aplicação de exemplo: a loja virtual

Como o foco do livro não é no processo de desenvolvimento em si, mas sim nas práticas de DevOps que auxiliam o *build*, *deploy* e operação de uma aplicação em produção, usaremos uma aplicação fictícia baseada em um projeto de código aberto. A loja virtual é uma aplicação web escrita em Java sobre a plataforma *Broadleaf Commerce* (http://www.broadleafcommerce.org/) . O código utilizado neste capítulo e no restante do livro foi escrito com base no site de demonstração oferecido pela Broadleaf e pode ser acessado no seguinte repositório do GitHub:

https://github.com/dtsato/loja-virtual-devops/

O *Broadleaf Commerce* é uma plataforma flexível que expõe pontos de configuração e de extensão que podem ser customizados para implementar funcionalidades específicas. No entanto, boa parte das funcionalidades padrão de um site de compras online como o Submarino ou a Amazon já estão disponíveis, incluindo:

- Navegação e busca no catálogo de produtos;
- Páginas de produto com nome, descrição, preço, fotos e produtos relacionados;
- Carrinho de compras;
- Processo de *checkout* customizável incluindo códigos promocionais, dados de cobrança e de entrega;
- Cadastro de usuário;
- Histórico de compras;
- Ferramentas de administração para: catálogo de produtos, promoções, preços, taxas de frete e páginas com conteúdo customizado.

Figura 2.1: Uma prévia da página inicial da loja virtual

Além disso, o *Broadleaf Commerce* utiliza diversos frameworks bem estabelecidos na comunidade Java, tornando o exemplo mais interessante do ponto de vista de DevOps, pois representa bem a complexidade envolvida no processo de *build* e *deploy* de uma aplicação Java no mundo real. Os detalhes de implementação vão além do escopo deste livro, porém é importante conhecer um pouco das principais tecnologias e frameworks utilizados nesta aplicação:

- **Java**: a aplicação é escrita em Java (http://java.oracle.com/) , compatível com a versão Java SE 6 e superiores.

- **Spring**: o Spring (http://www.springframework.org/) é um framework de desenvolvimento de aplicações corporativas popular na comunidade Java que oferece diversos componentes como: injeção de dependências, gerenciamento de transações, segurança, um framework de MVC, dentre outros.

- **JPA e Hibernate**: o JPA é a API de persistência do Java e o Hibernate (http://www.hibernate.org/) é a implementação mais famosa da JPA na comunidade Java, oferecendo recursos para realizar o mapeamento objeto-relacional (*object-relational mapping* ou ORM) entre objetos Java e as tabelas no banco de dados.

- **Google Web Toolkit**: o GWT (http://developers.google.com/web-toolkit/) é um framework desenvolvido pelo Google para facilitar a criação de interfaces ricas que rodam no browser. O GWT permite que o desenvolvedor escreva código Java que é então compilado para Javascript. A loja virtual utiliza o GWT para implementar a interface gráfica das ferramentas de administração.
- **Apache Solr**: o Solr (http://lucene.apache.org/solr/) é um servidor de pesquisa que permite a indexação do catálogo de produtos da loja virtual e oferece uma API eficiente e flexível para efetuar consultas de texto em todo o catálogo.
- **Tomcat**: o Tomcat (http://tomcat.apache.org/) é um servidor que implementa as tecnologias web – Java Servlet e JavaServer Pages – do Java EE. Apesar do *Broadleaf Commerce* rodar em servidores de aplicação alternativos – como Jetty, GlassFish ou JBoss – utilizaremos o Tomcat por ser uma escolha comum em diversas empresas rodando aplicações web Java.
- **MySQL**: o MySQL (http://www.mysql.com/) é um servidor de banco de dados relacional. O JPA e o Hibernate permitem que a aplicação rode em diversos outros servidores de banco de dados – como Oracle, PostgreSQL ou SQL Server – porém também utilizaremos o MySQL por ser uma escolha popular e por disponibilizar uma versão de código aberto.

Não é uma aplicação pequena. Melhor ainda: ela utiliza bibliotecas que são frequentemente encontradas na grande maioria das aplicações Java no mercado. Como falamos, vamos utilizá-la como nosso exemplo, mas você também pode seguir o processo com a sua própria aplicação, ou ainda escolher outro software, independente de linguagem, para aprender as técnicas de DevOps que serão apresentadas e discutidas durante o livro.

O próximo objetivo é colocar a loja virtual no ar, porém, antes de fazer o primeiro *deploy*, é preciso ter um ambiente de produção pronto, com servidores onde o código possa rodar. O ambiente de produção em nosso exemplo será inicialmente composto de 2 servidores, conforme mostra a figura 2.2.
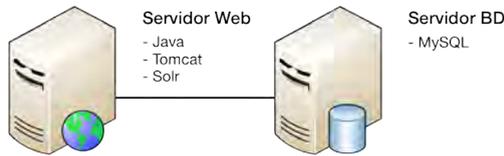
Figura 2.2: Ambiente de produção da loja virtual

Usuários acessarão a loja virtual, que rodará em uma instância do Tomcat no servidor web. O servidor web rodará todas as bibliotecas e frameworks Java utilizados pela loja virtual, inclusive uma instância embutida do Solr. Por fim, a aplicação web utilizará o MySQL, rodando em um servidor de banco de dados separado. Esta é uma arquitetura web de duas camadas comumente utilizada por diversas aplicações no mundo real. No capítulo 8 discutiremos com mais detalhes os fatores que influenciam a escolha da arquitetura física para sua aplicação, porém por enquanto vamos usar algo comum para simplificar o exemplo e colocar a loja virtual no ar o quanto antes.

2.2 Instalando o ambiente de produção

Comprar servidores e hardware para a loja virtual custaria muito dinheiro, então inicialmente iremos usar máquinas virtuais. Isso permitirá criar todo o ambiente de produção na nossa máquina. Existem diversas ferramentas para rodar máquinas virtuais – como VMware ou Parallels – porém algumas são pagas e a maioria usa uma interface gráfica para configuração, o que tornaria este capítulo uma coleção de *screenshots* difíceis de acompanhar.

Para resolver esses problemas, usaremos duas ferramentas que facilitam o uso e configuração de máquinas virtuais: o **Vagrant** (http://www.vagrantup.com) e o **VirtualBox** (http://www.virtualbox.org) . O VirtualBox é uma ferramenta da Oracle que permite configurar e rodar máquinas virtuais nas principais plataformas: Windows, Linux, Mac OS X e Solaris. O VirtualBox possui uma ferramenta de linha de comando para executar e configurar as máquinas virtuais, porém para facilitar ainda mais o trabalho, usaremos o Vagrant que fornece uma DSL em Ruby para definir, gerenciar e configurar ambientes virtuais. O Vagrant possui também uma interface de linha de comando simples

para subir e interagir com esses ambientes virtuais.

Caso você já possua o Vagrant e o VirtualBox instalados e funcionando, você pode pular para a subseção *"Declarando e Subindo os Servidores"*. Caso contrário, o processo de instalação dessas ferramentas é simples.

Instalando o VirtualBox

Para instalar o VirtualBox, acesse a página de *download* http://www.virtualbox.org/wiki/Downloads e escolha a versão mais atual. No momento da escrita do livro, a versão mais nova é o VirtualBox 4.3.8 e será a versão utilizada no decorrer do livro. Escolha o pacote de instalação de acordo com a sua plataforma: no Windows o instalador é um arquivo executável `.exe`; no Mac OS X o instalador é um pacote `.dmg`; no Linux o VirtualBox disponibiliza pacotes `.deb` ou `.rpm`, dependendo da sua distribuição.

Uma vez que você tenha feito o *download*, instale o pacote. No Windows e no Mac OS X, basta dar um duplo clique no arquivo de instalação (`.exe` no Windows e `.dmg` no Mac OS X) e seguir as instruções do instalador. No Linux, caso tenha escolhido o pacote `.deb`, instale-o executando o comando `dpkg -i {arquivo.deb}`, substituindo `{arquivo.deb}` pelo nome do arquivo baixado, por exemplo `virtualbox-4.3_4.3.8-92456~Ubuntu~raring_i386.deb`. Caso tenha escolhido o pacote `.rpm`, instale-o executando o comando `rpm -i {arquivo.rpm}`, substituindo `{arquivo.rpm}` pelo nome do arquivo baixado, por exemplo `VirtualBox-4.3-4.3.8_92456_el6-1.i686.rpm`. Observação: no Linux, caso seu usuário não seja `root`, você precisará rodar os comandos anteriores com o comando `sudo` na frente, por exemplo: `sudo dpkg -i {arquivo.deb}` ou `sudo rpm -i {arquivo.rpm}`.

Para testar que o VirtualBox está instalado corretamente, acesse o terminal e execute o comando `VBoxManage -v`. Para abrir o terminal no Windows, você pode usar `Win+R` e digitar `cmd`. Caso tudo esteja correto, o comando retornará algo como "4.3.8r92456", dependendo da versão instalada.

Instalando o Vagrant

Uma vez que o VirtualBox estiver instalado, prossiga com o processo de instalação do Vagrant, que é bem parecido. Acesse a página de *download* do

Vagrant http://www.vagrantup.com/downloads.html e escolha a versão mais atual. No momento da escrita do livro, a versão mais nova é o Vagrant 1.5.1 e será a versão utilizada no decorrer do livro. Faça o *download* do pacote de instalação de acordo com a sua plataforma: no Windows o instalador é um arquivo `.msi`; no Mac OS X o instalador é um pacote `.dmg`; no Linux o Vagrant disponibiliza pacotes `.deb` ou `.rpm`, dependendo da sua distribuição.

Assim que tenha feito o *download* do pacote, instale-o: no Windows e no Mac OS X, basta dar um duplo clique no arquivo de instalação (`.msi` no Windows e `.dmg` no Mac OS X) e seguir as instruções do instalador. No Linux, basta seguir os mesmos passos da instalação do VirtualBox no Linux usando o pacote escolhido (`vagrant_1.5.1_i686.deb` ou `vagrant_1.5.1_i686.rpm`). No Mac OS X e no Windows, o comando `vagrant` já é colocado no `PATH` após a instalação. No Linux, você vai precisar adicionar `/opt/vagrant/bin` no seu `PATH`.

Para testar que o Vagrant está instalado corretamente, acesse o terminal e execute o comando `vagrant -v`. Caso tudo esteja correto, o comando retornará algo como "Vagrant 1.5.1", dependendo da versão instalada.

O último passo necessário é configurar uma imagem inicial que serve de *template* para inicializar cada máquina virtual. Estas imagens, também conhecidas como *box*, servem como ponto de partida, contendo o sistema operacional básico. No nosso caso, utilizaremos uma *box* oferecida pelo próprio Vagrant, que contém a imagem de um Linux Ubuntu 12.04 LTS de 32 bits. Para baixar e configurar esta *box*, é preciso executar o comando:

```
$ vagrant box add hashicorp/precise32
==> box: Loading metadata for box 'hashicorp/precise32'
    box: URL: https://vagrantcloud.com/hashicorp/precise32
...
```

Esse comando irá fazer o *download* do arquivo de imagem da máquina virtual, que é grande (299MB), portanto ele vai demorar para executar e uma boa conexão com a internet será necessária. Uma lista de outras imagens disponibilizadas pela comunidade Vagrant pode ser encontrada em http://www.vagrantbox.es/ e o processo de inicialização é parecido com o comando anterior. A empresa por trás do Vagrant, a HashiCorp, introduziu

recentemente o **Vagrant Cloud** (https://vagrantcloud.com/) , uma forma de encontrar e compartilhar *boxes* com a comunidade.

Declarando e subindo os servidores

Uma vez que o Vagrant e o VirtualBox estão funcionando, declarar e gerenciar um ambiente virtual é simples. A declaração de configuração das máquinas virtuais é feita em um arquivo chamado `Vagrantfile`. O comando `vagrant init` cria um arquivo `Vagrantfile` inicial, com comentários que explicam todas as opções de configuração disponíveis.

No nosso caso, iremos começar de forma simples e configurar apenas o necessário para subir os dois servidores de produção. O conteúdo do `Vagrantfile` que precisamos é:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
config.vm.box = "hashicorp/precise32"

config.vm.define :db do |db_config|
  db_config.vm.network :private_network, :ip => "192.168.33.10"
end

config.vm.define :web do |web_config|
  web_config.vm.network :private_network, :ip => "192.168.33.12"
end
end
```

Esse arquivo configura duas máquinas virtuais, com apelidos de " `db`" e " `web`". Ambas utilizarão a *box* `hashicorp/precise32` que instalamos anteriormente. Cada uma possui um bloco de configuração que define um endereço IP para configurar a conexão com a rede. Os endereços IP `192.168.33.10` e `192.168.33.12` foram escolhidos de forma arbitrária para evitar conflitos com as configurações de rede da sua máquina.

Para subir os servidores, basta executar o comando `vagrant up` no mesmo diretório onde o arquivo `Vagrantfile` foi criado e o Vagrant tomará conta de subir e configurar as máquinas virtuais para você. Se tudo correr bem, a saída do comando será algo parecido com:

```
$ vagrant up
Bringing machine 'db' up with 'virtualbox' provider...
Bringing machine 'web' up with 'virtualbox' provider...
==> db: Importing base box 'hashicorp/precise32'...
==> db: Matching MAC address for NAT networking...
==> db: Checking if box 'hashicorp/precise32' is up to date...
==> db: Setting the name of the VM: blank_db_1394680068450_7763
==> db: Clearing any previously set network interfaces...
==> db: Preparing network interfaces based on configuration...
    db: Adapter 1: nat
    db: Adapter 2: hostonly
==> db: Forwarding ports...
    db: 22 => 2222 (adapter 1)
==> db: Booting VM...
==> db: Waiting for machine to boot. This may take a few
                                                    minutes...
    db: SSH address: 127.0.0.1:2222
    db: SSH username: vagrant
    db: SSH auth method: private key
==> db: Machine booted and ready!
==> db: Configuring and enabling network interfaces...
==> db: Mounting shared folders...
    db: /vagrant => /private/tmp/blank
==> web: Importing base box 'hashicorp/precise32'...
==> web: Matching MAC address for NAT networking...
==> web: Checking if box 'hashicorp/precise32' is up to date...
==> web: Setting the name of the VM:
                                    blank_web_1394680087001_3842
==> web: Fixed port collision for 22 => 2222. Now on port 2200.
==> web: Clearing any previously set network interfaces...
==> web: Preparing network interfaces based on configuration...
    web: Adapter 1: nat
    web: Adapter 2: hostonly
==> web: Forwarding ports...
    web: 22 => 2200 (adapter 1)
==> web: Booting VM...
==> web: Waiting for machine to boot. This may take a few
                                                    minutes...
    web: SSH address: 127.0.0.1:2200
```

```
    web: SSH username: vagrant
    web: SSH auth method: private key
==> web: Machine booted and ready!
==> web: Configuring and enabling network interfaces...
==> web: Mounting shared folders...
    web: /vagrant => /private/tmp/blank
```

Isso significa que as máquinas virtuais estão funcionando e prontas para serem usadas. Se você ler a saída com cuidado, verá que os passos foram os mesmos para ambas as máquinas `db` e `web`. Além disso, o Vagrant também mapeou portas de SSH e montou uma partição para acessar os arquivos existentes no diretório da máquina hospedeira. Isso pode soar um pouco confuso, mas demonstra algumas das funcionalidades poderosas que o Vagrant disponibiliza.

2.3 Configurando dependências nos servidores de produção

Agora que os servidores estão rodando, é preciso instalar os pacotes e dependências para que a loja virtual funcione. Administradores de sistema provavelmente já estão acostumados a trabalhar em diversas máquinas ao mesmo tempo, porém desenvolvedores geralmente trabalham sempre em uma máquina só.

Nas próximas seções começaremos a interagir com mais de uma máquina ao mesmo tempo e a maior parte dos comandos precisarão ser executados no servidor correto. Usaremos uma notação especial para guiar as instruções no restante do livro, representando o usuário e o servidor no *prompt* da linha de comando para servir de lembrete para você verificar se está rodando o comando no servidor certo. O padrão será `{usuário}@{servidor}$`, no qual o usuário geralmente será `vagrant` e o servidor pode variar entre `db` ou `web`. Desta forma, comandos precedidos de `vagrant@db$` devem ser executados no servidor `db` e comandos precedidos de `vagrant@web$` devem ser executados no servidor `web`.

Ao invés de pular de um servidor para o outro, iremos focar inicialmente em um servidor por vez, começando pelo servidor de banco de dados.

Servidor de banco de dados

Primeiramente, precisamos logar no servidor de banco de dados para instalar o MySQL. Isso é feito através do comando `vagrant ssh db`, que abre uma sessão SSH com o servidor `db`. Uma vez logado, a instalação do servidor MySQL é feita através do pacote `mysql-server`:

```
vagrant@db$ sudo apt-get update
Ign http://us.archive.ubuntu.com precise InRelease
Ign http://us.archive.ubuntu.com precise-updates InRelease
...
Reading package lists... Done
vagrant@db$ sudo apt-get install mysql-server
Reading package lists... Done
Building dependency tree
...
ldconfig deferred processing now taking place
```

Para entender o que está acontecendo, vamos analisar cada parte desse comando. O comando `apt-get` é usado para gerenciar quais pacotes estão instalados no sistema. A instrução `update` atualiza o índice de pacotes para as últimas versões. A instrução `install` serve para instalar um novo pacote no sistema. Nesse caso o pacote que queremos instalar é o pacote `mysql-server` que contém o servidor MySQL. O comando inteiro é precedido pelo comando `sudo`. Em sistemas UNIX, existe um usuário especial com "super poderes" conhecido como usuário `root` e o comando `sudo` permite executar algo como se o usuário atual fosse `root`.

O processo de instalação do servidor MySQL exige a escolha de uma senha de acesso para o usuário `root`. Você precisará fornecer essa senha duas vezes no processo de instalação. A escolha de uma senha segura é importante para garantir a segurança do sistema, porém é também um tópico de discussão à parte, que abordaremos no capítulo 8. Se alguém descobrir a senha mestre, ele poderá executar qualquer comando no banco de dados, inclusive apagar todas as tabelas! Na vida real você não quer que isso aconteça, porém, no exemplo do livro, usaremos uma senha simples para facilitar: "secret". Se preferir, você pode escolher outra senha, mas daqui para frente tome cuidado

para substituir qualquer menção da senha "secret" pela senha que você escolheu.

Ao terminar a instalação do pacote, o servidor MySQL já vai estar rodando, porém apenas para acesso local. Como precisamos acessar o banco de dados do servidor web, é preciso configurar o MySQL para permitir conexões externas. Para isto, basta criar o arquivo de configuração `/etc/mysql/conf.d/allow_external.cnf`. Você pode usar o editor de texto que preferir – `emacs` e `vim` são duas opções comuns – porém aqui usaremos o `nano` por ser um editor mais simples de aprender para quem não está acostumado a editar arquivos na linha de comando. O comando seguinte cria o arquivo (como `root` novamente) desejado:

```
vagrant@db$ sudo nano /etc/mysql/conf.d/allow_external.cnf
```

Basta então digitar o conteúdo do arquivo abaixo e usar `Ctrl+O` para salvar e `Ctrl+X` para salvar e sair do editor:

```
[mysqld]
  bind-address = 0.0.0.0
```

Uma vez que o arquivo foi criado, é preciso reiniciar o servidor MySQL para que ele perceba a nova configuração. Isso é feito com o comando:

```
vagrant@db$ sudo service mysql restart
mysql stop/waiting
mysql start/running, process 6460
```

Neste momento, o servidor MySQL já está pronto e rodando, porém ainda não criamos nenhum banco de dados. É comum que cada aplicação crie seu próprio banco de dados, ou *schema*, para persistir suas tabelas, índices e dados. Precisamos então criar um banco de dados para a loja virtual, que chamaremos de `loja_schema`, através do comando:

```
vagrant@db$ mysqladmin -u root -p create loja_schema
Enter password:
```

Ao executar este comando, a opção `-u root` instrui que a conexão com o MySQL deve ser feita como usuário root e a opção `-p` irá perguntar a senha, que nesse caso será "secret". O resto do comando é autoexplicativo. Para

verificar que o banco de dados foi criado com sucesso, você pode executar o seguinte comando que lista todos os banco de dados existentes:

```
vagrant@db$ mysql -u root -p -e "SHOW DATABASES"
Enter password:
+--------------------+
| Database           |
+--------------------+
| information_schema |
| loja_schema        |
| mysql              |
| performance_schema |
| test               |
+--------------------+
```

Assim como não é recomendável executar comandos no sistema operacional como usuário `root`, não é recomendável executar queries no MySQL como usuário `root`. Uma boa prática é criar usuários específicos para cada aplicação, liberando apenas o acesso necessário para a aplicação rodar. Além disso, a instalação padrão do servidor MySQL inclui uma conta anônima para acesso ao banco de dados. Antes de criar a conta para a loja virtual, precisamos remover a conta anônima:

```
vagrant@db$ mysql -uroot -p -e "DELETE FROM mysql.user WHERE
                                                    user=''; \
> FLUSH PRIVILEGES"
Enter password:
```

Uma vez que a conta anônima foi removida, criaremos um usuário chamado `loja`, com uma senha "lojasecret", com acesso exclusivo ao *schema* `loja_schema`:

```
vagrant@db$ mysql -uroot -p -e "GRANT ALL PRIVILEGES ON
                                               loja_schema.* \
> TO 'loja'@'%' IDENTIFIED BY 'lojasecret';"
Enter password:
```

Para testar que o novo usuário foi criado corretamente, podemos executar uma *query* simples para verificar que tudo está OK (a senha deve ser "lojase-

cret" pois estamos executando o comando como usuário `loja` ao invés de `root`):

```
vagrant@db$ mysql -u loja -p loja_schema -e
  "select database(), user()"
Enter password:
+--------------+----------------+
| database()   | user()         |
+--------------+----------------+
| loja_schema  | loja@localhost |
+--------------+----------------+
```

Neste ponto, o servidor de banco de dados já está configurado e rodando corretamente. Para deslogar da máquina virtual `db`, você pode digitar o comando `logout` ou pressionar `Ctrl+D`. O banco de dados ainda está vazio – nenhuma tabela ou dados foram criados – porém iremos cuidar disto na seção 2.4. Antes disso, precisamos terminar de configurar o resto da infraestrutura: o servidor web.

Servidor web

Iremos instalar o Tomcat e configurar uma fonte de dados para usar o banco de dados criado anteriormente. Para isto, precisamos logar no servidor rodando o comando `vagrant ssh web`, que abre uma sessão SSH com o servidor `web`. Uma vez logado, a instalação do Tomcat é feita através do pacote `tomcat7`. Também precisamos instalar o cliente para conectar com o banco de dados MySQL, disponível no pacote `mysql-client`:

```
vagrant@web$ sudo apt-get update
Ign http://us.archive.ubuntu.com precise InRelease
Ign http://us.archive.ubuntu.com precise-updates InRelease
...
Reading package lists... Done
vagrant@web$ sudo apt-get install tomcat7 mysql-client
Reading package lists... Done
Building dependency tree
...
ldconfig deferred processing now taking place
```

Isto irá instalar o Tomcat 7, o cliente MySQL e todas as suas dependências, incluindo o Java 6. Para verificar que o Tomcat está rodando corretamente, você pode abrir seu navegador e digitar a URL http://192.168.33.12:8080/ do servidor web – `192.168.33.12` foi o endereço IP configurado no Vagrant e `8080` é a porta onde o Tomcat roda por padrão. Se tudo estiver ok, você deve ver uma página com uma mensagem *"It works!"*.

A loja virtual precisa de uma conexão segura com o servidor para transferir dados pessoais – como senhas ou números de cartão de crédito – de forma criptografada. Para isso, precisamos configurar um conector SSL para que o Tomcat consiga rodar tanto em HTTP quanto HTTPS. Conexões SSL usam um certificado assinado por uma autoridade confiável para identificar o servidor e para criptografar os dados que trafegam entre o seu navegador e o servidor. Usaremos a ferramenta `keytool`, que é parte da distribuição padrão do Java, para gerar um certificado para o nosso servidor web. Durante este processo, você precisará fornecer diversas informações sobre o servidor além de uma senha para proteger o `keystore`, que no nosso caso será novamente "secret" para simplificar. O comando completo, juntamente com exemplos dos dados que precisam ser fornecidos, fica:

```
vagrant@web$ cd /var/lib/tomcat7/conf
vagrant@web$ sudo keytool -genkey -alias tomcat -keyalg RSA \
> -keystore .keystore
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Loja Virtual
What is the name of your organizational unit?
  [Unknown]:  Devops na Prática
What is the name of your organization?
  [Unknown]:  Casa do Código
What is the name of your City or Locality?
  [Unknown]:  São Paulo
What is the name of your State or Province?
  [Unknown]:  SP
What is the two-letter country code for this unit?
  [Unknown]:  BR
Is CN=Loja Virtual, OU=Devops na Prática, ..., C=BR correct?
```

```
  [no]:  yes

Enter key password for <tomcat>
  (RETURN if same as keystore password):
```

Explicando o comando passo a passo: a opção `-genkey` gera uma nova chave; a opção `-alias tomcat` define um apelido para o certificado; a opção `-keyalg RSA` especifica o algoritmo usado para gerar a chave, nesse caso RSA; por fim, a opção `-keystore .keystore` define o arquivo que armazenará o `keystore`. Este formato de arquivo permite armazenar diversas chaves/certificados no mesmo `keystore`, por isso é bom escolher um apelido para o certificado gerado para o Tomcat. Além disso, tanto o `keystore` quanto o certificado do Tomcat são protegidos por senha. Nesse caso, usamos "secret" para ambos.

Uma vez que o certificado foi criado e armazenado, precisamos configurar o servidor do Tomcat para habilitar o conector SSL. Isso é feito editando o arquivo `/var/lib/tomcat7/conf/server.xml`:

```
vagrant@web$ sudo nano /var/lib/tomcat7/conf/server.xml
```

Dentro deste arquivo, a definição do conector SSL para a porta 8443 já está predefinida, porém comentada. Você precisa descomentar a definição deste conector e adicionar mais dois atributos que representam o arquivo do `keystore` e a senha de proteção. A parte relevante do arquivo XML que precisamos alterar é:

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
  ...
  <Service name="Catalina">
    ...
    <!-- Define a SSL HTTP/1.1 Connector on port 8443
     This connector uses the JSSE configuration, when using
     APR, the connector should be using the OpenSSL style
     configuration described in the APR documentation -->

    <Connector port="8443" protocol="HTTP/1.1"
               SSLEnabled="true"
```

```
               maxThreads="150" scheme="https" secure="true"
               keystoreFile="conf/.keystore"
               keystorePass="secret"
               clientAuth="false" sslProtocol="SSLv3" />
    ...
  </Service>
</Server>
```

A configuração do servidor web está quase pronta aqui. A última alteração que precisamos fazer é aumentar a quantidade de memória que o Tomcat pode usar. A loja virtual exige mais memória do que a instalação padrão do Tomcat define, por isso precisamos alterar uma linha no arquivo `/etc/default/tomcat7`:

```
vagrant@web$ sudo nano /etc/default/tomcat7
```

A linha que define a opção `JAVA_OPTS` precisa ser alterada para permitir que a máquina virtual do Java use até 512Mb de memória, ao invés da configuração padrão de 128Mb. Após a alteração, a opção `JAVA_OPTS` deve ficar:

```
JAVA_OPTS="-Djava.awt.headless=true -Xmx512M
                                    -XX:+UseConcMarkSweepGC"
```

Por fim, para que todas as configurações sejam aplicadas, precisamos reiniciar o servidor Tomcat:

```
vagrant@web$ sudo service tomcat7 restart
```

Para verificar que tudo está funcionando, você pode voltar no navegador e tentar acessar a página padrão através do conector SSL, acessando a URL https://192.168.33.12:8443/ . Dependendo de qual navegador você está utilizando, um aviso irá surgir para alertar que o certificado do servidor foi autoassinado. Em um cenário real, você geralmente compra um certificado SSL assinado por uma autoridade confiável, mas nesse caso iremos prosseguir e aceitar o alerta pois sabemos que o certificado foi autoassinado intencionalmente. Se tudo der certo, você verá novamente a página *"It works!"*, dessa vez usando uma conexão segura em HTTPS.

Neste momento, ambos os servidores estão configurados e prontos para a última etapa: fazer o *build* e *deploy* para colocar a loja virtual em produção!

2.4 Build e deploy da aplicação

Até agora, a configuração dos servidores de banco de dados e web foram relativamente genéricas. Instalamos os pacotes de software básicos para rodar o MySQL e o Tomcat, configuramos conectores para HTTP/HTTPS, criamos um usuário e um banco de dados vazio, que por enquanto não estão diretamente associados à loja virtual a não ser pelo nome do *schema* e do usuário que escolhemos.

Chegou o momento de baixar e compilar o código, rodar os testes, empacotar a aplicação e colocar a loja virtual no ar! Esse processo de compilação, teste e empacotamento é conhecido como *build*. As etapas do processo de *build* também podem incluir gerenciamento de dependências, rodar ferramentas de análise estática do código, cobertura de código, geração de documentação, dentre outros. A principal função do processo de *build* é gerar um ou mais artefatos, com versões específicas, que se tornam potenciais candidatos, ou *release candidates*, para o *deploy* em produção.

Geralmente recomenda-se rodar o processo de *build* em um ambiente parecido com o ambiente de produção para evitar incompatibilidades de sistema operacional ou versões de bibliotecas. Poderíamos até configurar o Vagrant para criar uma nova máquina virtual para utilizarmos como servidor de *build*. Porém, para facilitar a discussão neste capítulo, faremos o *build* no servidor web, que é onde o *deploy* vai acontecer. Isso economizará diversos passos agora, porém vamos revisitar este processo no capítulo 6.

Para realizar o *build*, é preciso instalar mais algumas ferramentas: o **Git** (http://git-scm.com/) para controle de versão, o **Maven** (http://maven.apache.org/) para executar o *build* em si e o **JDK** (*Java Development Kit*) para conseguirmos compilar o código Java:

```
vagrant@web$ sudo apt-get install git maven2 openjdk-6-jdk
Reading package lists... Done
Building dependency tree
...
ldconfig deferred processing now taking place
```

Uma vez que essas ferramentas foram instaladas, precisamos baixar o código que está hospedado no Github através do comando `git clone`, e executar o *build* rodando o comando `mvn install`:

```
vagrant@web$ cd
vagrant@web$ git clone https://github.com/dtsato/
                                      loja-virtual-devops.git
Cloning into 'loja-virtual-devops'...
remote: Counting objects: 11358, done.
remote: Compressing objects: 100% (3543/3543), done.
remote: Total 11358 (delta 6053), reused 11358 (delta 6053)
Receiving objects: 100% (11358/11358), 55.47 MiB | 1.14 MiB/s,
                                                         done.
Resolving deltas: 100% (6053/6053), done.
vagrant@web$ cd loja-virtual-devops
vagrant@web$ export MAVEN_OPTS=-Xmx256m
vagrant@web$ mvn install
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   loja-virtual-devops
[INFO]   core
[INFO]   admin
[INFO]   site
[INFO]   combined
[INFO] ----------------------------------------------------
[INFO] Building loja-virtual-devops
[INFO]    task-segment: [install]
[INFO] ----------------------------------------------------
...
```

A primeira execução do Maven vai demorar de 20 a 25 minutos pois diversas coisas acontecem:

- **Sub-projetos**: o projeto está organizado em 4 módulos: `core` possui as configurações principais, extensões e customizações da loja virtual; `site` é a aplicação web da loja virtual; `admin` é uma aplicação web restrita para administração e gerenciamento da loja virtual; por fim, `combined` é um módulo que agrega as duas aplicações web e o `core` em um único artefato. Cada módulo atua como um subprojeto que o Maven precisa realizar o *build* separadamente.
- **Resolução de dependências**: o Maven irá resolver todas as dependências do projeto e baixar as bibliotecas e frameworks necessários para

compilar e rodar cada módulo da loja virtual.

- **Compilação**: o código Java e GWT precisa ser compilado para execução. Isso toma um bom tempo na primeira vez. O compilador Java é inteligente o suficiente para recompilar apenas o necessário em *builds* subsequentes.

- **Testes automatizados**: em módulos que definem testes automatizados, o Maven irá executá-los e gerar relatórios de quais testes passaram e quais falharam.

- **Empacotamento de artefatos**: por fim, o módulo `core` será empacotado como um arquivo `.jar` enquanto os módulos web (`site`, `admin` e `combined`) serão empacotados como um arquivo `.war`. Ambos os arquivos `jar` e `war` são equivalentes a arquivos `zip` que você baixa regularmente da internet, no entanto a estrutura interna dos pacotes é bem definida e padronizada pelo Java: o `jar` contém classes compiladas e serve como biblioteca enquanto o `war` contém classes, bibliotecas, arquivos estáticos e de configuração necessários para rodar uma aplicação web em um *container* Java como o Tomcat.

Ao final da execução do processo de *build*, se tudo correr bem, você deve ver um relatório do Maven parecido com:

```
...
[INFO] ----------------------------------------------------
[INFO] Reactor Summary:
[INFO] ----------------------------------------------------
[INFO] loja-virtual-devops .............. SUCCESS [1:35.191s]
[INFO] core ............................. SUCCESS [6:51.591s]
[INFO] admin ............................ SUCCESS [6:54.377s]
[INFO] site ............................. SUCCESS [4:44.313s]
[INFO] combined ......................... SUCCESS [25.425s]
[INFO] ----------------------------------------------------
[INFO] ----------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ----------------------------------------------------
[INFO] Total time: 20 minutes 31 seconds
```

```
[INFO] Finished at: Sat Mar 15 02:34:24 UTC 2014
[INFO] Final Memory: 179M/452M
[INFO] ------------------------------------------------------------------------
```

Isso significa que o *build* rodou com sucesso e o artefato que nos interessa para *deploy* é o arquivo `devopsnapratica.war` do módulo `combined`, que se encontra no diretório `combined/target`.

Antes de fazer o *deploy* do arquivo `war`, a última configuração necessária no Tomcat é a definição das fontes de dado, ou `DataSource`, que a aplicação irá acessar para conectar com o banco de dados. Se tais definições estivessem dentro do artefato em si, nós precisaríamos realizar um novo *build* toda vez que os parâmetros de conexão com o banco de dados mudassem ou caso quiséssemos rodar a mesma aplicação em ambientes diferentes. Em vez disso, a aplicação depende de recursos abstratos através da especificação JNDI (*Java Naming and Directory Interface*). Estes recursos precisam ser configurados e expostos pelo *container*, possibilitando a alteração sem necessidade de um novo *build*.

A loja virtual precisa de três fontes de dados no JNDI chamadas `jdbc/web`, `jdbc/secure` e `jdbc/storage`. No nosso caso, usaremos o mesmo banco de dados para os três recursos, definidos no arquivo `context.xml`:

```
vagrant@web$ sudo nano /var/lib/tomcat7/conf/context.xml
```

As configurações relevantes são os três elementos `<Resource>` e a diferença entre eles é apenas o atributo `name`:

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
  <WatchedResource>WEB-INF/web.xml</WatchedResource>

  <Resource name="jdbc/web" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000" username="loja" password="lojasecret"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://192.168.33.10:3306/loja_schema"/>
```

```
  <Resource name="jdbc/secure" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000" username="loja" password="lojasecret"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://192.168.33.10:3306/loja_schema"/>

  <Resource name="jdbc/storage" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000" username="loja" password="lojasecret"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://192.168.33.10:3306/loja_schema"/>
</Context>
```

Para fazer o *deploy* da loja virtual em si, simplesmente precisamos copiar o artefato `war` para o lugar certo e o Tomcat fará o necessário para colocar a loja virtual no ar:

```
vagrant@web$ cd ~/loja-virtual-devops
vagrant@web$ sudo cp combined/target/devopsnapratica.war \
> /var/lib/tomcat7/webapps
```

Assim como o *build* demorou um pouco, o processo de *deploy* também vai levar alguns minutos na primeira vez, pois a aplicação irá criar as tabelas e popular os dados usando o Hibernate. Você pode acompanhar o processo de *deploy* olhando para os logs do Tomcat. O comando `tail -f` é útil nesse caso pois ele mostra em tempo real tudo que está acontecendo no log:

```
vagrant@web$ tail -f /var/lib/tomcat7/logs/catalina.out
Mar 15, 2014 2:08:51 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8080"]
Mar 15, 2014 2:08:51 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8443"]
Mar 15, 2014 2:08:51 AM org.apache.catalina.startup.Catalina
                                                           start
INFO: Server startup in 408 ms
...
```

O *deploy* termina assim que você perceber que as atividades no log pararam. Para terminar o comando `tail` você pode usar `Ctrl+C`. Agora é só

testar que tudo está funcionando: abra uma nova janela no navegador e digite a URL http://192.168.33.12:8080/devopsnapratica/ . Parabéns, a loja virtual está em produção!
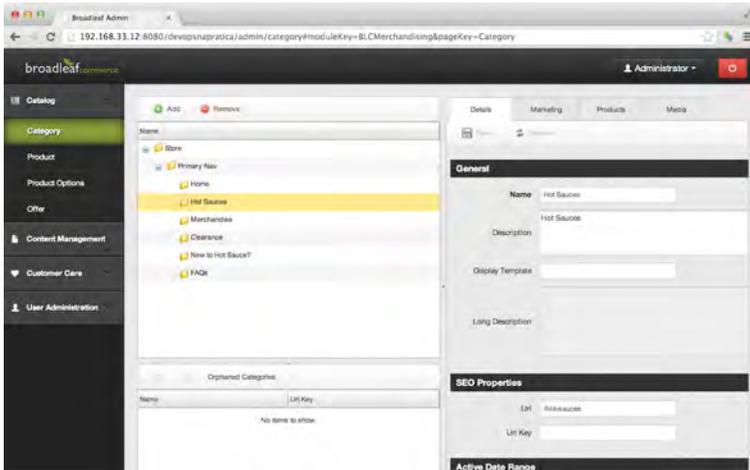
Figura 2.3: Página de administração da loja virtual

Você também pode acessar a página de administração que se encontra na URL http://192.168.33.12:8080/devopsnapratica/admin/ (a barra no final da URL é importante). Entrando com o usuário "admin" e senha "admin" você pode navegar pelas páginas de administração e gerenciar o catálogo de produtos, categorias, o conteúdo das páginas da loja virtual, gerenciar pedidos, assim como usuários e permissões.

Agora que estamos em produção e conquistamos a última milha, podemos focar nos princípios e práticas de DevOps necessários para operar nossa aplicação em produção, ao mesmo tempo permitindo que novas mudanças sejam introduzidas de forma confiável.

Capítulo 3

Monitoramento

Agora que a loja virtual está em produção, conquistamos a parte mais importante do ciclo de vida de qualquer *software*. No entanto, este foi apenas o primeiro passo! Daqui para frente, precisamos vestir o chapéu do time de operações e nos responsabilizar pela estabilidade do sistema em produção: precisamos garantir que a loja virtual continue funcionando sem problemas.

Como saber se o sistema está funcionando normalmente ou se está fora do ar? Você pode acessar a loja virtual manualmente e verificar se está tudo ok, porém isso leva tempo e depende de você lembrar de fazê-lo. Você também pode esperar por reclamações de seus usuários, mas essa também não é uma boa opção. Sistemas de monitoramento existem para resolver esse problema. Você define quais verificações são necessárias para saber se o sistema está funcionando e o sistema de monitoramento envia notificações assim que alguma verificação falhar.

Um bom sistema de monitoramento irá alertá-lo quando algo estiver er-

rado, permitindo que você investigue e arrume o que estiver causando problemas o mais rápido possível. Desta forma, você não precisa depender de seus usuários avisarem quando algo está fora do ar, possibilitando uma abordagem mais proativa para resolução de problemas.

Neste capítulo iremos instalar um sistema de monitoramento e configurá-lo com várias verificações para garantir que a loja virtual esteja funcionando em produção. Utilizaremos uma das ferramentas de monitoração mais utilizadas por equipes de operações: o **Nagios** (http://www.nagios.org/) . Apesar de ser uma ferramenta antiga e possuir uma interface gráfica não muito amigável, ele ainda é uma das opções mais populares por ter uma implementação robusta e um ecossistema enorme de plugins escritos e mantidos pela comunidade. Ao final do capítulo, nosso ambiente de produção incluirá um novo servidor, conforme mostra a figura 3.1.
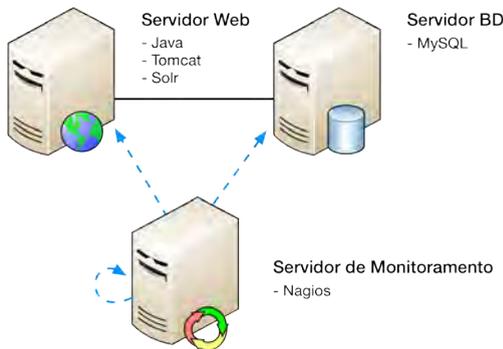
Figura 3.1: Ambiente de produção da loja virtual

3.1 Instalando o servidor de monitoramento

Em um sistema de monitoramento, os servidores geralmente atuam em dois papéis principais: *supervisor* ou *supervisionado*. O **supervisor** é responsável por coletar informações sobre o resto da infraestrutura, avaliar se ela está saudável e enviar alertas caso encontre algum problema. Os **supervisionados** são os servidores que fazem o trabalho real que você está interessado em monitorar. Na nossa loja virtual, o servidor web e o servidor de banco de dados

serão supervisionados por um novo servidor que iremos instalar e configurar para atuar como supervisor.

Rodar o supervisor externamente aos servidores supervisionados é uma boa prática. Se o supervisor estiver rodando na mesma máquina e algum problema acontecer, ambos ficarão indisponíveis e você não conseguirá ser alertado do problema. Por esse motivo, iremos declarar uma nova máquina virtual no arquivo `Vagrantfile`, com o apelido `monitor` e endereço IP `192.168.33.14`:

```
Vagrant.configure("2") do |config|
  config.vm.box = "precise32"

  config.vm.define :db do |db_config|
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
  end

  config.vm.define :web do |web_config|
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

Após salvar o arquivo, basta executar `vagrant up` no mesmo diretório para subir o novo servidor. Se tudo der certo, a saída do comando será parecida com:

```
$ vagrant up
Bringing machine 'db' up with 'virtualbox' provider...
Bringing machine 'web' up with 'virtualbox' provider...
Bringing machine 'monitor' up with 'virtualbox' provider...
==> db: Checking if box 'hashicorp/precise32' is up to date...
==> db: VirtualBox VM is already running.
```

```
==> web: Checking if box 'hashicorp/precise32' is up to date...
==> web: VirtualBox VM is already running.
==> monitor: Importing base box 'hashicorp/precise32'...
==> monitor: Matching MAC address for NAT networking...
==> monitor: Checking if box 'hashicorp/precise32' is
                                                   up to date...
==> monitor: Setting the name of the VM:
              blank_monitor_1394851889588_2267
==> monitor: Fixed port collision for 22 => 2222.
                                                   Now on port 2201.
==> monitor: Clearing any previously set network interfaces...
==> monitor: Preparing network interfaces based on
                                                   configuration...
    monitor: Adapter 1: nat
    monitor: Adapter 2: hostonly
==> monitor: Forwarding ports...
    monitor: 22 => 2201 (adapter 1)
==> monitor: Running 'pre-boot' VM customizations...
==> monitor: Booting VM...
==> monitor: Waiting for machine to boot. This may take a few
                                                         minutes...
    monitor: SSH address: 127.0.0.1:2201
    monitor: SSH username: vagrant
    monitor: SSH auth method: private key
    monitor: Error: Connection timout. Retrying...
==> monitor: Machine booted and ready!
==> monitor: Configuring and enabling network interfaces...
==> monitor: Mounting shared folders...
    monitor: /vagrant => /private/tmp/blank
```

Para completar a instalação básica, precisamos logar no servidor usando o comando `vagrant ssh monitor`. Uma vez logado, a instalação do Nagios é feita através do pacote `nagios3`. Porém, antes de instalar o Nagios, precisamos configurar o sistema para usar a última versão. Para isto, precisamos executar os seguintes comandos:

```
vagrant@monitor$ echo "Package: nagios*
> Pin: release n=raring
> Pin-Priority: 990" | sudo tee /etc/apt/preferences.d/nagios
```

```
Package: nagios*
Pin: release n=raring
Pin-Priority: 990
vagrant@monitor$ echo "deb http://archive.ubuntu.com/ubuntu \
> raring main" | sudo tee /etc/apt/sources.list.d/raring.list
deb http://archive.ubuntu.com/ubuntu raring main
vagrant@monitor$ sudo apt-get update
Ign http://archive.ubuntu.com raring InRelease
Ign http://security.ubuntu.com precise-security InRelease
...
```

Isso adiciona uma nova entrada na lista de pacotes com o *release* mais novo do Ubuntu – chamado "raring" – e configura as preferências do instalador de pacotes para dar prioridade ao novo *release* nos pacotes do Nagios. O comando `apt-get update` faz com que o instalador de pacotes atualize seu *cache* para saber das novas versões disponíveis. Uma vez feito isso, podemos instalar o nagios através do comando:

```
vagrant@monitor$ sudo apt-get install nagios3
Reading package lists... Done
Building dependency tree
...
```

O Nagios possui uma dependência com o Postfix, um programa UNIX responsável por envio de e-mails. Por esse motivo, durante a instalação irá aparecer uma tela rosa pedindo configurações para o Postfix. Escolha *Internet Site* para seguir o processo de instalação e na próxima tela, onde ele pede o nome do sistema para envio de e-mails, digite "monitor.lojavirtualdevops.com.br".

A próxima tela de configuração é do Nagios, e você precisa escolher uma senha para o usuário administrador "nagiosadmin" conseguir acessar a interface web. Usaremos a senha "secret". Após estas configurações, o processo de instalação deve terminar e você pode testar se a instalação funcionou abrindo o seu navegador web e visitando o endereço http://nagiosadmin:secret@192.168.33.14/nagios3 . Caso você tenha escolhido outro usuário ou senha, modifique a URL com os dados de acesso corretos. Se tudo der certo, você verá a interface web de administração do Nagios. Clicando no link *Services*, você verá uma tela parecida com a figura 3.2.
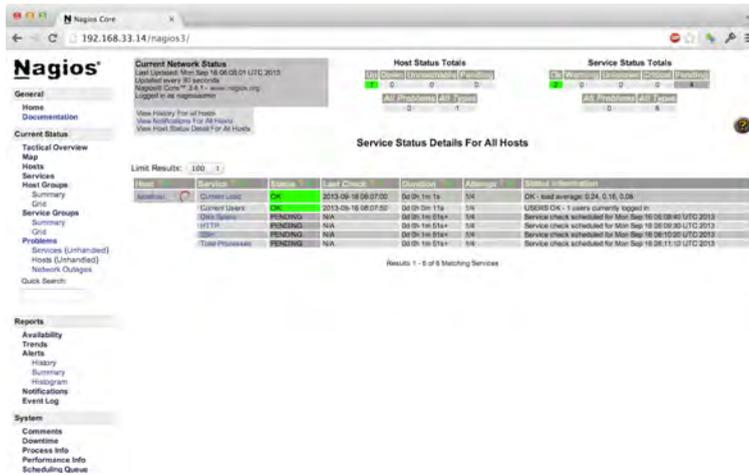
Figura 3.2: Interface de administração do Nagios

Como você pode perceber, algumas verificações padrão – ou serviços, na terminologia do Nagios – já estão pré-configuradas. Por enquanto, o servidor está apenas se automonitorando, atuando tanto como supervisor quanto supervisionado. Nesta tela, cada servidor monitorado – ou *host* – possui um conjunto de serviços. O servidor do Nagios está representado pelo nome *localhost* e cada linha na tabela representa uma verificação independente. O status verde "OK" significa que a verificação passou, enquanto o status "PENDING" significa que a verificação ainda não rodou. As verificações padrão são:

- **Carga atual**: a carga de um sistema UNIX mede o quão ocupada a CPU está. Para ser mais exato, a carga é uma média móvel do tamanho da fila de processos esperando para executar. Quando a CPU começa a se sobrecarregar, os processos precisam esperar mais tempo para rodar, aumentando a fila e fazendo o sistema rodar mais devagar. Esta verificação envia um alerta quando a carga atual ultrapassa um certo limite.
- **Usuários atuais**: verifica quantos usuários estão logados no sistema.

Essa é uma maneira simplificada de detectar intrusão no servidor e enviar alertas quando o número de usuários logados vai além do esperado.

- **Espaço de disco**: quando um servidor fica sem espaço em disco, diversos problemas começam a aparecer. Um dos motivos mais comuns para isso é a geração de logs em sistemas de produção. Para evitar este problema é importante monitorar quanto espaço livre ainda resta. Esta verificação envia alertas quando o espaço livre atinge um certo limite.
- **HTTP**: verifica que o servidor está aceitando conexões HTTP. Neste caso, o Nagios instalou um servidor web para que possamos acessar a interface de administração da figura 3.2. Por padrão, esta verificação enviará alertas caso não consiga abrir uma conexão HTTP na porta 80.
- **SSH**: verifica que o servidor está aceitando conexões SSH na porta 22 e envia alertas quando encontra algum problema.
- **Número total de processos**: outra forma de avaliar se o servidor está sobrecarregado é através do número total de processos. Se muitos processos estiverem executando ao mesmo tempo, o sistema operacional vai gastar mais tempo gerenciando qual pode rodar na CPU, não sobrando tempo suficiente para que nenhum deles rode efetivamente. Esta verificação envia alertas quando o número total de processos ultrapassa um certo limite.

A primeira vista, pode parecer que são muitas verificações para monitorar um único servidor. No entanto, seguindo o mesmo princípio de escrever poucas asserções por teste quando desenvolvemos código usando TDD, verificações mais granulares ajudam a identificar o ponto de falha mais rapidamente quando algum problema acontece. Em vez de receber um alerta genérico dizendo que o servidor está com problemas, você vai ter informações específicas para identificar a causa do problema mais rapidamente.

Nas próximas seções discutiremos um pouco mais sobre os conceitos e as configurações do Nagios e aprenderemos como supervisionar outros servidores, como adicionar mais verificações e como receber alertas quando algo dá errado.

3.2 Monitorando outros servidores

Com o Nagios instalado e efetuando verificações, precisamos configurá-lo para monitorar nossos outros servidores de produção. Para isto precisamos declarar onde estão os outros servidores e quais verificações executar. Criaremos um novo arquivo para definir todas as configurações pertinentes à loja virtual:

```
vagrant@monitor$ sudo nano /etc/nagios3/conf.d/loja_virtual.cfg
```

Neste arquivo iremos declarar os servidores da loja virtual, definindo novos *hosts* com os respectivos endereços IP:

```
define host {
  use         generic-host
  host_name   192.168.33.10
  hostgroups  ssh-servers, debian-servers
}

define host {
  use         generic-host
  host_name   192.168.33.12
  hostgroups  ssh-servers, debian-servers
}
```

Sempre que alteramos um arquivo de configuração, precisamos recarregá-los para que o Nagios detecte as mudanças. Para isto, usamos o comando `reload`:

```
vagrant@monitor$ sudo service nagios3 reload
```

No Nagios, um *host* representa um dispositivo a ser monitorado, como por exemplo um servidor, um roteador ou uma impressora. Podemos também agrupar *hosts* semelhantes em *host groups* e definir configurações para diversos servidores de uma vez, evitando duplicação. O mesmo *host* pode ser membro de mais de um *host group*.

No arquivo de configuração `/etc/nagios3/conf.d/loja_virtual.cfg`, definimos que os nossos servidores são membros de dois grupos:

`ssh-servers` e `debian-servers`. O grupo `ssh-servers` cria uma verificação para o serviço SSH. O grupo `debian-servers` não define nenhuma verificação extra, mas adiciona informações como o logotipo da Debian que aparece no ícone do servidor na interface web do Nagios.

Para verificar que as configurações foram aplicadas com sucesso, você pode recarregar a página de administração e esperar até que as novas verificações SSH estejam OK, conforme mostra a figura 3.3.
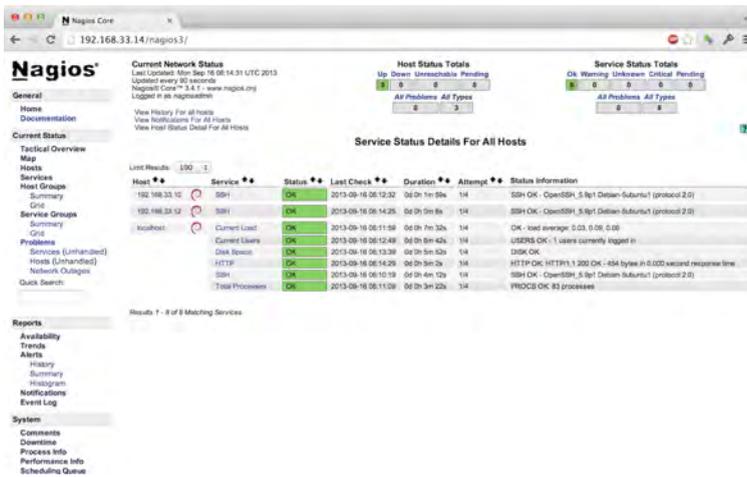
Figura 3.3: Monitorando serviço SSH em mais servidores

Para facilitar a configuração, iremos adicionar nossos servidores a dois novos *host groups*: um grupo chamado `web-servers` para os servidores web e um grupo chamado `db-servers` para os servidores de banco de dados. Faremos isso editando o mesmo arquivo de configuração:

```
vagrant@monitor$ sudo nano /etc/nagios3/conf.d/loja_virtual.cfg
```

Será preciso adicionar os novos *host groups* e modificar a configuração dos *hosts* existentes para adicioná-los aos grupos corretos:

```
define hostgroup {
  hostgroup_name   db-servers
  alias            Database Servers
```

```
}

define hostgroup {
  hostgroup_name  web-servers
  alias           Web Servers
}

define host {
  use         generic-host
  host_name   192.168.33.10
  hostgroups  ssh-servers, debian-servers, db-servers
}

define host {
  use         generic-host
  host_name   192.168.33.12
  hostgroups  ssh-servers, debian-servers, web-servers
}
```

Após recarregar o arquivo de configuração – usando o mesmo comando `sudo service nagios3 reload` – você pode verificar que os novos grupos existem clicando no link *Host Groups* na tela de administração do Nagios.

Executar diversas verificações e enviar alertas quando algo está errado são as funcionalidades mais importantes de um sistema de monitoramento como o Nagios. Na próxima seção, vamos conhecer os diversos tipos de checagens que estão disponíveis para verificar que a loja virtual está funcionando corretamente.

3.3 Explorando os comandos de verificação do Nagios

Cada serviço do Nagios está associado a um comando que é executado em cada verificação. O comando nada mais é do que um script que pode ser executado diretamente da linha de comando. Diversos comandos já vêm instalados com o Nagios por padrão, porém você pode criar seus próprios scripts ou instalar plugins da comunidade para estender sua biblioteca de verificações.

Vamos explorar alguns dos comandos pré-instalados, que estão disponíveis no diretório `/usr/lib/nagios/plugins`. Por exemplo, o comando `check_ssh`, que é executado na verificação do serviço SSH:

```
vagrant@monitor$ cd /usr/lib/nagios/plugins
vagrant@monitor$ ./check_ssh 192.168.33.12
SSH OK - OpenSSH_5.9p1 Debian-5ubuntu1 (protocol 2.0)
vagrant@monitor$ echo $?
0
vagrant@monitor$ ./check_ssh 192.168.33.11
No route to host
vagrant@monitor$ echo $?
2
```

Perceba que ao rodar o comando `check_ssh` passando o endereço IP do servidor web, ele imprime uma mensagem com um resumo de uma linha do que aconteceu. Essa mensagem geralmente é da forma **<VERIFICAÇÃO> <STATUS> - <MENSAGEM COM DETALHES>**, porém o que define o resultado da verificação não é o formato da mensagem, mas sim, o código de saída do processo – ou *exit code*, representado pela variável `$?` no bash:

- **0 – OK**: indica que a verificação passou com sucesso. Representado pelo status verde na interface web.
- **1 – WARNING**: indica que a verificação passou o primeiro limite de alerta. Representado pelo status amarelo na interface web.
- **2 – CRITICAL**: indica que a verificação falhou e algo está errado. Representado pelo status vermelho na interface web.
- **3 – UNKNOWN**: indica que a verificação não conseguiu decidir se o serviço está bem ou não. Representado pelo status laranja na interface web.

Outra verificação que podemos fazer é que conseguimos acessar o servidor web por HTTP:

```
vagrant@monitor$ ./check_http -H 192.168.33.12
Connection refused
```

```
HTTP CRITICAL - Unable to open TCP socket
vagrant@monitor$ echo $?
2
vagrant@monitor$ ./check_http -H 192.168.33.12 -p 8080
HTTP OK: HTTP/1.1 200 OK - 2134 bytes in 0.007 second response
                                                         time ...
vagrant@monitor$ echo $?
0
```

Perceba que precisamos passar a opção `-H` para indicar qual *host* deve ser verificado. Além disso, como não especificamos a porta a ser verificada, o comando `check_http` usa a porta 80 por padrão. Ao passarmos a opção `-p` com o valor 8080 – a porta onde o servidor Tomcat está escutando – a verificação passa com sucesso.

Um último comando para explorarmos é a verificação de quanto espaço em disco ainda está disponível:

```
vagrant@monitor$ ./check_disk -H 192.168.33.12 -w 10% -c 5%
/usr/lib/nagios/plugins/check_disk: invalid option -- 'H'
Unknown argument
Usage:
 check_disk -w limit -c limit [-W limit] [-K limit] ...
vagrant@monitor$ ./check_disk -w 10% -c 5%
DISK OK - free space: / 74473 MB (97% inode=99%); ...
```

Perceba que, ao contrário do comando `check_http`, o comando `check_disk` não aceita a opção `-H` e consegue apenas verificar quanto espaço em disco está disponível no servidor local. Isso acontece pois o serviço HTTP é um serviço de rede acessível remotamente. Por outro lado, o comando `check_disk` precisa de informações locais do servidor para decidir quanto espaço de disco está disponível e não pode ser executado remotamente.

O Nagios possui três opções quando você precisa realizar uma verificação remota de um serviço que não está disponível na rede:

- **Checagem por SSH**: através do comando `check_by_ssh`, o Nagios consegue abrir uma conexão SSH com o servidor remoto e executar o

comando lá. Esta é uma opção simples pois não exige a instalação de nenhum agente no servidor supervisionado. No entanto, conforme o número de verificações remotas aumenta, isso pode aumentar a carga no servidor supervisor, pois ele precisará criar e destruir diversas conexões SSH.

- **Checagem ativa com NRPE**: o *Nagios Remote Plugin Executor*, ou NRPE, é um plugin que permite a execução de verificações que dependem de recursos locais no servidor supervisionado. Um processo agente do NRPE roda no servidor supervisionado e fica esperando pedidos de verificação do supervisor. Quando o Nagios precisa executar uma verificação, ele pede para o agente NRPE rodá-la e coleta os resultados quando a verificação termina. Este tipo de verificação é ativa pois quem inicia o pedido de execução do comando é o servidor supervisor.

- **Checagem passiva com NSCA**: o *Nagios Service Check Acceptor*, ou NSCA, é um plugin que permite a execução de verificações remotas de forma passiva no servidor supervisionado. Um processo cliente do NSCA roda no servidor supervisionado e executa as verificações periodicamente, enviando os resultados para um servidor NSCA que roda no supervisor. Este tipo de verificação é passiva pois quem inicia a execução do comando é o próprio servidor supervisionado.

A figura 3.4 mostra as diferentes formas de realizar verificações com o Nagios. Verificações de serviços de rede, como `check_http` rodam direto do servidor supervisor. Verificações que precisam rodar no servidor supervisionado, podem ser chamadas ativamente com `check_by_ssh` ou através do plugin NRPE. Por fim, verificações passivas podem rodar de forma independente no servidor supervisionado e os resultados enviados para o Nagios através do plugin NSCA.
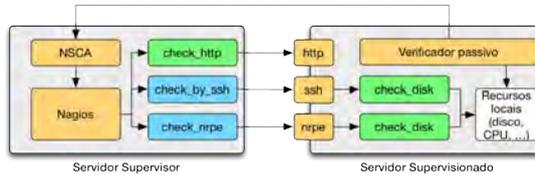
Figura 3.4: Verificações remotas com Nagios

Agora que já conhecemos os diferentes tipos de verificações do Nagios e sabemos como executar seus diferentes comandos, vamos tornar o monitoramento da infraestrutura da loja virtual mais robusto.

3.4 Adicionando verificações mais específicas

Além das verificações padrão, você pode declarar seus próprios serviços e associá-los a um único *host* ou a um grupo de *hosts*. Na infraestrutura da loja virtual, o servidor de banco de dados está rodando o MySQL como um serviço de rede. Podemos verificar que o serviço está disponível adicionando uma verificação associada ao grupo `db-servers` no final do arquivo de configuração `/etc/nagios3/conf.d/loja_virtual.cfg`:

```
define hostgroup ...
define host ...

define service {
  service_description  MySQL
  use                  generic-service
  hostgroup_name       db-servers
  check_command        check_tcp!3306
}
```

Após recarregar o Nagios e esperar pela próxima execução da verificação, um novo serviço "MySQL" aparecerá com o status OK. Esta verificação usa o comando `check_tcp` que tenta abrir uma conexão TCP na porta especificada – nesse caso a porta onde o MySQL roda é 3306. A sintaxe do Nagios para passar argumentos para um comando é separá-los com um ponto de exclamação (`!`).

Esta verificação garante que existe algum processo aceitando conexões TCP na porta 3306, porém não garante que tal processo seja o MySQL. Para verificar que o banco de dados realmente está lá, é preciso usar um outro comando `check_mysql`, já instalado no diretório de plugins `/usr/lib/nagios/plugins`. Executando diretamente da linha de comando, podemos ver quais argumentos ele aceita:

```
vagrant@monitor$ /usr/lib/nagios/plugins/check_mysql --help
...
Usage:
  check_mysql [-d database] [-H host] [-P port] [-s socket]
        [-u user] [-p password] [-S]
...
```

Vamos então adicionar uma nova verificação no arquivo de configuração da loja virtual:

```
define hostgroup ...
define host ...
define service ...

define service {
  service_description  MySQL-lojavirtual
  use                  generic-service
  hostgroup_name       db-servers
  check_command        check_mysql_database!loja!lojasecret
                                                   !loja_schema
}
```

Você deve ter percebido que o comando executado na linha de comando se chama `check_mysql`, porém o comando que usamos para declarar o serviço se chama `check_mysql_database`. Isso acontece pois os comandos precisam ser declarados dentro do Nagios antes de serem usados por um serviço. Nesse caso, o plugin já definiu os comandos do Nagios e podemos ver suas declarações olhando o conteúdo do arquivo `/etc/nagios-plugins/config/mysql.cfg`:

```
vagrant@monitor$ cat /etc/nagios-plugins/config/mysql.cfg
define command{
```

```
  command_name  check_mysql
  command_line  /usr/lib/nagios/plugins/check_mysql -H
                                                      '$HOSTADDRESS$'
}

...

define command{
  command_name  check_mysql_database
  command_line  /usr/lib/nagios/plugins/check_mysql -d '$ARG3$'
                                                              -H ...
}

...
```

O último comando permite passar argumentos para o nome de usuário, senha e qual banco de dados queremos verificar. Podemos ver também a ordem em que os argumentos devem ser passados quando declaramos nosso serviço: o Nagios substitui os tokens `$ARG1$`, `$ARG2$`, `$ARG3$` pelos valores separados pelo ponto de exclamação na respectiva ordem.

Agora que temos nosso banco de dados monitorado, é hora de adicionar algumas verificações para os servidores web. Seguindo o mesmo raciocínio de antes, os serviços de rede expostos pelo servidor web são as conexões HTTP e HTTPS do Tomcat. O comando que podemos usar para realizar tais verificações é o `check_http`, também instalado no diretório de plugins `/usr/lib/nagios/plugins`. Executando diretamente da linha de comando, podemos ver quais argumentos ele aceita:

```
vagrant@monitor$ /usr/lib/nagios/plugins/check_http --help
...
Usage:
 check_http -H <vhost> | -I <IP-address> [-u <uri>] [-p <port>]
  [-w <warn time>] [-c <critical time>] [-t <timeout>] [-L]
  [-a auth] [-b proxy_auth]
  [-f <ok|warning|critcal|follow|sticky|stickyport>]
  [-e <expect>] [-s string] [-l]
  [-r <regex> | -R <ignorecase regex>]
  [-P string] [-m <min_pg_size>:<max_pg_size>] [-4|-6] [-N]
```

```
  [-M <age>] [-A string] [-k string] [-S <version>] [--sni]
  [-C <warn_age>[,<crit_age>]] [-T <content-type>] [-j method]
NOTE: One or both of -H and -I must be specified
...
```

Nesse caso, apesar do plugin já declarar alguns comandos do Nagios no arquivo `/etc/nagios-plugins/config/http.cfg`, nenhum deles aceita o argumento `-p` para passar a porta. Como o tomcat não está rodando na porta HTTP padrão (80), antes de associarmos um novo serviço ao grupo de servidores `web-servers` precisamos declarar nossos próprios comandos. Faremos isso no arquivo de configuração da loja virtual (lembrando que a barra invertida indica apenas que o comando deve ser definido todo na mesma linha):

```
define hostgroup ...
define host ...
define service ...
define service ...

define command {
 command_name  check_tomcat_http
 command_line  /usr/lib/nagios/plugins/check_http -H
  '$HOSTADDRESS$' \ -p '$ARG1$' -u '$ARG2$' -e 'HTTP/1.1 200 OK'
}

define service {
  service_description Tomcat
  use                 generic-service
  hostgroup_name      web-servers
  check_command       check_tomcat_http!8080!'/devopsnapratica/'
}
```

O argumento `-H` representa o endereço do *host* monitorado, o primeiro argumento representa a porta da verificação HTTP e o segundo argumento é a URI. Nosso comando também passa a opção `-e` para garantir que a resposta do servidor é um status de sucesso: `HTTP/1.1 200 OK`.

Por fim, vamos adicionar uma última verificação para garantir que a conexão SSL também funciona na porta 8443 do Tomcat:

```
define hostgroup ...
define host ...
define service ...
define service ...
define command ...
define service ...

define command {
  command_name  check_tomcat_https
  command_line  /usr/lib/nagios/plugins/check_http -H
    '$HOSTADDRESS$' \ --ssl=3 -p '$ARG1$' -u '$ARG2$' -e
    'HTTP/1.1 200 OK'
}

define service {
  service_description Tomcat SSL
  use                 generic-service
  hostgroup_name      web-servers
  check_command       check_tomcat_https!8443!'/devopsnapratica/
                                                          admin/'
}
```

Após recarregar a configuração do Nagios e esperar pela execução das novas verificações, a tela de serviços na interface de administração deve estar parecida com a figura 3.5.
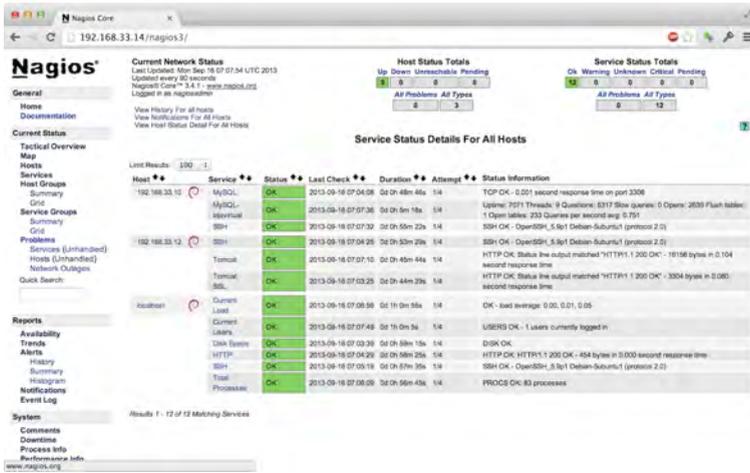
Figura 3.5: Todas as verificações da loja virtual

O arquivo de configuração `/etc/nagios3/conf.d/loja_virtual.cfg` completo, ao final desta seção ficará assim:

```
define hostgroup {
  hostgroup_name  db-servers
  alias           Database Servers
}

define hostgroup {
  hostgroup_name  web-servers
  alias           Web Servers
}

define host {
  use             generic-host
  host_name       192.168.33.10
  hostgroups      ssh-servers, debian-servers, db-servers
}

define host {
  use             generic-host
  host_name       192.168.33.12
```

```
  hostgroups      ssh-servers, debian-servers, web-servers
}

define service {
  service_description  MySQL
  use                  generic-service
  hostgroup_name       db-servers
  check_command        check_tcp!3306
}

define service {
  service_description  MySQL-lojavirtual
  use                  generic-service
  hostgroup_name       db-servers
  check_command        check_mysql_database!loja!lojasecret
                                                !loja_schema
}

define command {
command_name  check_tomcat_http
command_line  /usr/lib/nagios/plugins/check_http -H
  '$HOSTADDRESS$' \ -p '$ARG1$' -u '$ARG2$' -e 'HTTP/1.1 200 OK'
}

define service {
  service_description Tomcat
  use                 generic-service
  hostgroup_name      web-servers
  check_command       check_tomcat_http!8080!'/devopsnapratica/'
}

define command {
  command_name  check_tomcat_https
  command_line  /usr/lib/nagios/plugins/check_http -H
    '$HOSTADDRESS$' \ --ssl=3 -p '$ARG1$' -u '$ARG2$' -e
    'HTTP/1.1 200 OK'
}

define service {
```

```
  service_description Tomcat SSL
  use                 generic-service
  hostgroup_name      web-servers
  check_command       check_tomcat_https!8443!'/devopsnapratica/
                                                                admin/'
}
```

Agora que temos verificações mais detalhadas sobre o funcionamento da loja virtual, a última coisa que precisamos configurar para ter um sistema de monitoramento robusto é o recebimento de alertas.

3.5 Recebendo alertas

Ter verificações detalhadas do seu sistema é apenas o primeiro passo para construir um sistema de monitoramento robusto. Não adianta nada seu sistema de monitoramento detectar um problema se você não puder ser avisado. Por isso, é importante configurarmos alertas para receber informações sobre problemas o quanto antes possível.

Existem vários plugins do Nagios para enviar alertas de diversas maneiras: e-mail, pager, mensagem SMS, sistemas de chat como MSN, ICQ, Yahoo, ou qualquer coisa que você puder fazer da linha de comando. Você pode inclusive definir seu próprio comando para escolher como o Nagios enviará alertas. Por padrão, o Nagios já vem configurado com dois comandos para enviar alertas por e-mail: um deles notifica que o *host* está com problemas e o outro notifica que algum serviço está com problemas. Podemos ver a definição desses comandos no arquivo `/etc/nagios3/commands.cfg`:

```
vagrant@monitor$ cat /etc/nagios3/commands.cfg
...
define command {
  command_name  notify-host-by-email
  command_line  /usr/bin/printf "%b" ... | /usr/bin/mail -s ...
}

define command {
  command_name  notify-service-by-email
  command_line  /usr/bin/printf "%b" ... | /usr/bin/mail -s ...
```

```
}
...
```

Além desses comandos, existem mais dois objetos do Nagios que podem ser configurados para decidir quem recebe os alertas e quando: **contatos** e **períodos de tempo** – ou *time periods*. Períodos de tempo definem quando alertas podem ser enviados e estão definidos no arquivo `/etc/nagios3/conf.d/timeperiods_nagios2.cfg`. Olhando o conteúdo do arquivo podemos ver as diferentes opções, inclusive o famoso período de suporte 24x7 – 24 horas por dia, 7 dias por semana:

```
vagrant@monitor$ cat /etc/nagios3/conf.d/timeperiods_nagios2.cfg
...
define timeperiod {
  timeperiod_name 24x7
  alias           24 Hours A Day, 7 Days A Week
  sunday          00:00-24:00
  monday          00:00-24:00
  tuesday         00:00-24:00
  wednesday       00:00-24:00
  thursday        00:00-24:00
  friday          00:00-24:00
  saturday        00:00-24:00
}
...
```

Contatos são as pessoas que recebem alertas. No nosso caso, usaremos o único contato definido no Nagios para o usuário `root`, no arquivo `/etc/nagios3/conf.d/contacts_nagios2.cfg`:

```
vagrant@monitor$ cat /etc/nagios3/conf.d/contacts_nagios2.cfg
...
define contact {
  contact_name                  root
  alias                         Root
  service_notification_period   24x7
  host_notification_period      24x7
  service_notification_options  w,u,c,r
  host_notification_options     d,r
```

```
  service_notification_commands   notify-service-by-email
  host_notification_commands      notify-host-by-email
  email                           root@localhost
}
...
```

Na definição do contato você escolhe o período em que o contato estará em alerta, seu endereço de e-mail, os comandos executados para enviar alertas e as opções de notificação. Neste caso, seremos notificados quando o *host* estiver indisponível (`d` para *down*) ou recuperado (`r` para *recovery*) e quando serviços entrarem em status de alerta (`w` para *warning*), desconhecido (`u` para *unkown*), crítico (`c` para *critical*) ou recuperado (`r` para *recovery*). O Nagios também suporta configurações para grupos de contatos quando você quer alertar um time inteiro ao invés de uma única pessoa. No entanto, não precisaremos usar esta funcionalidade neste capítulo.

Para garantir que nossa infraestrutura consiga enviar alertas de verdade, precisamos escolher qual serviço de mensagens será usado pelo Nagios. Apesar de ninguém gostar de ser acordado de madrugada, times de operações precisam dar suporte 24x7 para seus sistemas e portanto preferem ser notificados por pager ou SMS. No entanto, é difícil encontrar algum serviço grátis para envio de SMS, especialmente para operadoras brasileiras. Algumas opções de serviços online pagos com cobertura SMS internacional são o **Twilio** (http://www.twilio.com/) e o **Nexmo** (http://www.nexmo.com/) .

Outro serviço online pago bastante usado é o **PagerDuty** (http://www.pagerduty.com/) por permitir gerenciar grupos, contatos, agenda de quem está *on-call* e formas de envio de alertas num único lugar, independente do Nagios. O PagerDuty suporta envio de SMS, e-mail, ou até notificações para iPhone ou Android. Antes de começar a pagar pelo plano mensal, é possível criar uma conta de teste válida por 30 dias e o PagerDuty possui uma página bem detalhada explicando como integrá-lo com o Nagios.

No nosso caso, para evitar uma dependência em um sistema externo pago, vamos usar a configuração padrão do Nagios de envio de alertas por e-mail. A única configuração necessária é criar um contato no Nagios para receber os alertas da loja virtual. Faremos isso editando o arquivo de configuração de contatos `/etc/nagios3/conf.d/contacts_nagios2.cfg`, trocando o

endereço de e-mail `root@localhost` pelo seu endereço de e-mail de verdade:

```
...
define contact {
  contact_name  root
  ...
  email         <seu_email>@gmail.com
}
...
```

Para que esta configuração tenha efeito, precisamos recarregar o serviço do Nagios pela última vez:

```
vagrant@monitor$ sudo service nagios3 reload
 * Reloading nagios3 monitoring daemon configuration files
 nagios3 ...done.
```

A partir de agora, qualquer alerta será enviado diretamente para o seu e-mail. Para testar que o envio de alertas está funcionando corretamente, você pode parar o serviço SSH e esperar a execução das próximas verificações do Nagios:

```
vagrant@monitor$ sudo service ssh stop
ssh stop/waiting
```

O Nagios irá tentar executar a verificação quatro vezes antes de declarar o serviço indisponível. Assim que o serviço aparecer com status vermelho na tela de administração do Nagios e o número de tentativas alcançar "4/4", você deverá receber um alerta por e-mail. Para que tudo volte ao normal, reinicie o serviço SSH:

```
vagrant@monitor$ sudo service ssh start
ssh start/running, process 3424
```

Assim que o Nagios detectar que o serviço voltou ao normal, você receberá um e-mail avisando que o serviço está recuperado.

3.6 Um problema atinge produção, e agora?

Agora que temos um sistema de monitoramento configurado para executar verificações e enviar alertas sobre eventuais problemas na infraestrutura da loja virtual, podemos simular um cenário de falha de um dos servidores. Essa também é uma boa maneira de testar se o sistema de monitoramento está funcionando corretamente.

Vamos imaginar que nosso servidor de banco de dados tenha sofrido um problema de hardware e parou de funcionar. Para isto, basta sair da máquina virtual de monitoramento executando o comando `logout` ou simplesmente digitando `CTRL-D` e pedir para o Vagrant destruir a máquina virtual do banco de dados:

```
vagrant@monitor$ logout
Connection to 127.0.0.1 closed.
$ vagrant destroy db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Forcing shutdown of VM...
==> db: Destroying VM and associated drives...
```

Dentro de alguns minutos, o sistema de monitoramento irá detectar que o servidor está indisponível e enviará um alerta. A partir deste momento, usuários não conseguem mais acessar a loja virtual para efetuar compras. Sendo o engenheiro responsável pela operação em produção, você recebe a notificação e responde prontamente para confirmar a falha e agora você precisa restituir o serviço.

Olhando o console do Nagios, você verá algo parecido com a figura 3.6. O *host* do banco de dados e todos os seus serviços estão vermelhos, porém a verificação do serviço HTTP no servidor web também falha, indicando que o problema do banco de dados está afetando a loja virtual.
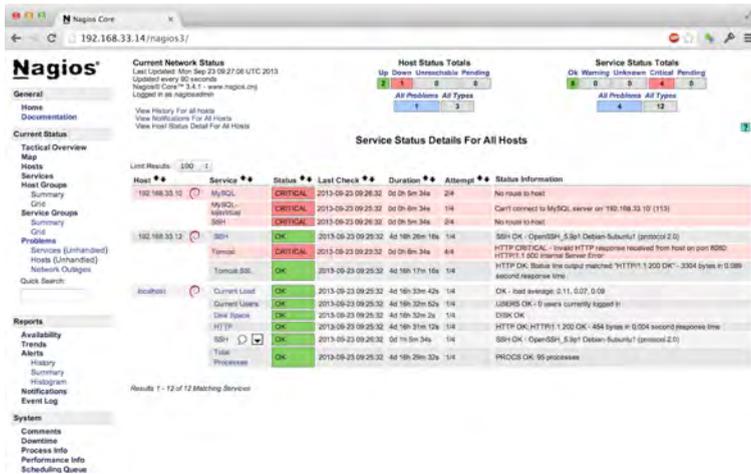
Figura 3.6: Verificações falhando quando um problema atinge produção

Para que tudo volte ao normal, você precisará reinstalar o servidor de banco de dados. No entanto, fizemos isto de forma manual no capítulo 2. Repetir todos aqueles passos manualmente levará muito tempo e, em momentos de crise como este, tempo é um recurso escasso. Será que não existe uma maneira melhor de instalar e configurar esses servidores?

Capítulo 4

Infraestrutura como código

Administradores de sistema aprendem o poder da linha de comando desde cedo e passam boa parte do dia no terminal controlando a infraestrutura da sua empresa. É comum dizer que eles estão alterando a **configuração do sistema**. Seja instalando novos servidores, instalando novo software, fazendo *upgrade* de software existente, editando arquivos de configuração, reiniciando processos, lendo arquivos de log, efetuando e restaurando *backups*, criando novas contas de usuário, gerenciando permissões, ajudando desenvolvedores a investigar algum problema ou escrevendo *scripts* para automatizar tarefas repetíveis.

Cada uma dessas tarefas altera ligeiramente o estado do servidor e pode potencialmente causar problemas se a mudança não for executada corretamente. É por isso que muitos administradores de sistema não dão acesso aos servidores de produção para desenvolvedores: para evitar mudanças indesejáveis e pela falta de confiança. No entanto, em muitas empresas, essas tarefas

são executadas manualmente nos servidores de produção pelos próprios administradores de sistema e eles também podem cometer erros.

Um dos principais princípios de DevOps é investir em automação. Automação permite executar tarefas mais rapidamente e diminui a possibilidade de erros humanos. Um processo automatizado é mais confiável e pode ser auditorado com mais facilidade. Conforme o número de servidores que precisam ser administrados aumenta, processos automatizados se tornam essenciais. Alguns administradores de sistema escrevem seus próprios *scripts* para automatizar as tarefas mais comuns. O problema é que cada um escreve sua própria versão dos *scripts* e fica difícil reutilizá-los em outras situações. Por outro lado, desenvolvedores são bons em escrever código modularizado e reutilizável.

Com o avanço da cultura DevOps e o aumento da colaboração entre administradores de sistema e desenvolvedores, diversas ferramentas têm evoluído para tentar padronizar o gerenciamento automatizado de infraestrutura. Tais ferramentas permitem tratar infraestrutura da mesma forma que tratamos código: usando controle de versões, realizando testes, empacotando e distribuindo módulos comuns e, obviamente, executando as mudanças de configuração no servidor.

Essa prática é conhecida como **infraestrutura como código** e nos permitirá resolver o problema que encontramos no final do capítulo 3. Em vez de reinstalar tudo manualmente, vamos usar uma ferramenta de gerenciamento de configurações para automatizar o processo de provisionamento, configuração e *deploy* da loja virtual.

4.1 Provisionamento, configuração ou deploy?

Ao comprar um laptop ou um celular novo, você só precisa ligá-lo e ele já está pronto para uso. Em vez de mandar as peças individuais e um manual de como montar sua máquina, o fabricante já fez todo o trabalho necessário para que você usufrua seu novo aparelho sem maiores problemas. Como usuário, você só precisa instalar programas e restaurar seus arquivos para considerar a máquina sua. Esse processo de preparação para o usuário final é conhecido como **provisionamento**.

O termo **provisionamento** é comumente usado por empresas de telecomunicações e por equipes de operações para se referir às etapas de preparação iniciais de configuração de um novo recurso: provisionar um aparelho celular, provisionar acesso à internet, provisionar um servidor, provisionar uma nova conta de usuário, e assim por diante. Provisionar um aparelho celular envolve, entre outras coisas: alocar uma nova linha, configurar os equipamentos de rede que permitem completar ligações, configurar serviços extra como SMS ou e-mail e, por fim, associar tudo isso com o chip que está no seu celular.

No caso de servidores, o processo de provisionamento varia de empresa para empresa dependendo da infraestrutura e da divisão de responsabilidades entre as equipes. Se a empresa possui infraestrutura própria, o processo de provisionamento envolve a compra e instalação física do novo servidor no seu *data center*. Se a empresa possui uma infraestrutura virtualizada, o processo de provisionamento só precisa alocar uma nova máquina virtual para o servidor. Da mesma forma, se a equipe de operações considerar a equipe de desenvolvimento como "usuário final", o processo de provisionamento acaba quando o servidor está acessível na rede, mesmo se a aplicação em si ainda não estiver rodando. Se considerarmos os verdadeiros usuários como "usuários finais", então o processo de provisionamento só acaba quando o servidor e a aplicação estiverem rodando e acessíveis na rede.

Para evitar maior confusão e manter integridade no decorrer do livro, vale a pena olhar para o processo de ponta a ponta e definir uma terminologia adequada. Imaginando o cenário mais longo de uma empresa que possui infraestrutura própria porém não tem servidores sobrando para uso, as etapas necessárias para colocar uma nova aplicação no ar seriam:

1) **Compra de hardware**: em empresas grandes essa etapa inicia um processo de aquisição, que envolve diversas justificações e aprovações pois investir dinheiro em hardware influi na contabilidade e no planejamento financeiro da empresa.

2) **Instalação física do hardware**: essa etapa envolve montar o novo servidor num *rack* no *data center*, assim como instalação de cabos de força, de rede etc.

3) **Instalação e configuração do sistema operacional**: uma vez que o servidor é ligado, é preciso instalar um sistema operacional e configurar os items básicos de hardware como: interfaces de rede, armazenamento (disco, partições, volumes em rede), autenticação e autorização de usuários, senha de *root*, repositório de pacotes etc.

4) **Instalação e configuração de serviços comuns**: além das configurações base do sistema operacional, muitos servidores precisam configurar serviços de infraestrutura básicos como: DNS, NTP, SSH, coleta e rotação de logs, backups, firewall, impressão etc.

5) **Instalação e configuração da aplicação**: por fim, é preciso instalar e configurar tudo que fará este servidor ser diferente dos outros: componentes de middleware, a aplicação em si, assim como configurações do middleware e da aplicação.

No decorrer do livro, usaremos o termo **provisionamento** para se referir ao processo que envolve as etapas 1 - 4, ou seja, todas as atividades necessárias para que o servidor possa ser usado e independente da razão pela qual ele foi requisitado. Usaremos o termo **deploy**, ou o equivalente em português **implantação**, quando nos referirmos à etapa 5. Apesar do *deploy* precisar de um servidor provisionado, o ciclo de vida do servidor é diferente da aplicação. O mesmo servidor pode ser usado por várias aplicações e cada aplicação pode efetuar *deploy* dezenas ou até centenas de vezes, enquanto o provisionamento de novos servidores acontece com menos frequência.

No pior caso, o processo de provisionamento pode demorar semanas ou até meses dependendo da empresa. Por esse motivo, é comum ver equipes de desenvolvimento definindo com bastante antecedência os requisitos de hardware para seus ambientes de produção e iniciando esse processo com a equipe de operações nas etapas iniciais do projeto, para evitar atrasos quando a aplicação estiver pronta para ir ao ar. Alguma colaboração pode acontecer no começo do processo, porém a maior parte do tempo as duas equipes trabalham em paralelo sem saber o que a outra está fazendo.

Pior ainda, muitas dessas decisões arquiteturais cruciais sobre a quantidade de servidores, configurações de hardware ou escolha do sistema operacional são tomadas no começo do projeto, no momento em que se tem a

menor quantidade de informação e conhecimento sobre as reais necessidades do sistema.

Um dos aspectos mais importantes da cultura DevOps é reconhecer esse conflito de objetivos e criar um ambiente de colaboração entre as equipes de desenvolvimento e de operações. O compartilhamento de práticas e ferramentas permite que a arquitetura se adapte e evolua conforme as equipes aprendem mais sobre o sistema rodando no mundo real: em produção. Esta visão holística também ajuda a encontrar soluções que simplifiquem o processo de compra e provisionamento, removendo etapas ou utilizando automação para fazer com que elas aconteçam mais rápido.

Um bom exemplo dessa simplificação do processo é o uso de tecnologias como virtualização de hardware e computação em nuvem – também conhecida pelo termo em inglês *cloud computing* ou simplesmente *cloud*. Provedores de serviço *cloud* de ponta permitem que você tenha um novo servidor no ar em questão de minutos através de um simples clique de botão ou chamada de API!

4.2 Ferramentas de gerenciamento de configuração

Na seção anterior, explicamos a diferença entre provisionamento e *deploy*. Com exceção dos items 1 e 2 que envolvem compra e instalação física de hardware, as outras três etapas envolvem algum tipo de configuração do sistema: seja do sistema operacional em si, do software base instalado na máquina ou da aplicação que roda em cima dele. A figura 4.1 mostra as diferentes etapas do ciclo de vida de um servidor e o gerenciamento de configuração atual.
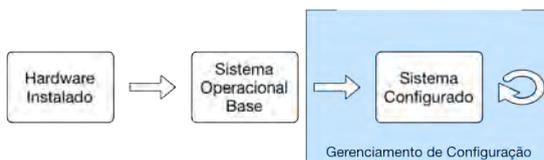
Figura 4.1: Escopo do gerenciamento de configuração

Existem diversas ferramentas para gerenciamento de configuração: ***shell scripts*** (a linha de comando também é conhecida como *shell*), **Puppet** (http://www.puppetlabs.com/puppet/) , **Chef** (http://www.opscode.com/chef/) , **Ansible** (http://www.ansibleworks.com/tech/) , **Salt** (http://saltstack.com/community) , **Pallet** (http://palletops.com/) e **CFEngine** (http://cfengine.com/) são algumas das mais populares.

A maioria dessas ferramentas possui uma empresa responsável pelo seu desenvolvimento e pelo crescimento da comunidade e ecossistema ao seu redor. Todas elas estão contribuindo para a evolução da comunidade DevOps e, apesar de competirem entre si, seu maior adversário ainda é a falta de uso de ferramentas. Por incrível que pareça, a maioria das empresas ainda gerenciam seus servidores de forma manual ou possuem *scripts* proprietários escritos há tanto tempo que ninguém consegue dar manutenção.

A grande diferença entre *shell scripts* proprietários e as outras ferramentas é que geralmente o *script* só funciona para instalar e configurar o servidor pela primeira vez. Ele serve como documentação executável dos passos executados na primeira instalação, porém se o *script* for executado novamente, ele provavelmente não vai funcionar. O administrador de sistemas precisa ser bem disciplinado e precisa escrever código extra para lidar com as situações onde algum pacote já está instalado ou precisa ser removido ou atualizado.

As outras ferramentas especializadas têm uma característica importante, conhecida como **idempotência**: você pode executá-las diversas vezes seguidas com o mesmo código e elas mudarão apenas o que for necessário. Se um pacote já está instalado, ele não será reinstalado. Se um serviço já está rodando, ele não será reiniciado. Se o arquivo de configuração já possui o conteúdo correto, ele não será alterado.

Idempotência permite escrever código de infraestrutura de forma declarativa. Ao invés da instrução ser *"instale o pacote X"* ou *"crie o usuário Y"*, você diz *"eu quero que o pacote X esteja instalado"* ou *"eu quero que o usuário Y exista"*. Você declara o estado final desejado e quando a ferramenta executa, caso o pacote ou o usuário já existam, nada irá acontecer.

A próxima coisa que você pode estar se perguntando é *"Qual dessas ferramentas devo usar?"*. Neste capítulo, usaremos o Puppet por permitir exemplificar bem essa propriedade declarativa da linguagem, por ser uma ferramenta

madura, aceita na comunidade de administradores de sistema e por ser bem adotada na comunidade. Porém, a verdade é que usar qualquer uma dessas ferramentas é melhor que nada. As ferramentas nesse espaço estão evoluindo constantemente e é importante você se familiarizar com algumas delas antes de tomar a decisão de qual delas é melhor para sua situação.

4.3 Introdução ao Puppet: recursos, provedores, manifestos e dependências

Cada uma das ferramentas de gerenciamento de configuração possui uma terminologia própria para se referir aos elementos da linguagem e aos componentes do seu ecossistema. Em um nível fundamental, cada comando que você declara na linguagem é uma **diretiva** e você pode definir um conjunto delas em um **arquivo de diretivas**, que é o equivalente a um arquivo de código-fonte de uma linguagem de programação como Java ou Ruby.

No Puppet, essas diretivas são chamadas de **recursos**. Alguns exemplos de recursos que você pode declarar no Puppet são: pacotes, arquivos, usuários, grupos, serviços, trechos de *script* executável, dentre outros. O arquivo no qual você declara um conjunto de diretivas é chamado de **manifesto**. Ao escrever código Puppet você passará a maior parte do tempo criando e editando esses manifestos.

Para se familiarizar com a sintaxe da linguagem Puppet e seu modelo de execução, vamos reconstruir a máquina virtual do servidor de banco de dados e logar por SSH na máquina que está zerada:

```
$ vagrant up db
Bringing machine 'db' up with 'virtualbox' provider...
==> db: Importing base box 'hashicorp/precise32'...
==> db: Matching MAC address for NAT networking...
==> db: Checking if box 'hashicorp/precise32' is up to date...
==> db: Setting the name of the VM: blank_db_1394854529922_29194
==> db: Clearing any previously set network interfaces...
==> db: Preparing network interfaces based on configuration...
    db: Adapter 1: nat
    db: Adapter 2: hostonly
==> db: Forwarding ports...
```

```
    db: 22 => 2222 (adapter 1)
==> db: Running 'pre-boot' VM customizations...
==> db: Booting VM...
==> db: Waiting for machine to boot. This may take a few
        minutes...
    db: SSH address: 127.0.0.1:2222
    db: SSH username: vagrant
    db: SSH auth method: private key
==> db: Machine booted and ready!
==> db: Configuring and enabling network interfaces...
==> db: Mounting shared folders...
    db: /vagrant => /private/tmp/blank
$ vagrant ssh db
Welcome to Ubuntu 12.04 LTS (GNU/Linux ...
vagrant@db$
```

Aproveitando que a *box* do Vagrant já possui o Puppet instalado, podemos começar a usá-lo diretamente criando um novo arquivo de manifesto vazio chamado `db.pp`. Por padrão, manifestos possuem extensão `.pp`:

```
vagrant@db$ nano db.pp
```

Como conteúdo inicial iremos declarar um recurso para o pacote `mysql-server` e dizer que o queremos instalado no sistema. Isso é feito passando o valor `installed` para o parâmetro `ensure` do recurso `package`:

```
package { "mysql-server":
  ensure => installed,
}
```

Se você comparar essa declaração de recurso com o comando que executamos no capítulo 2 – `sudo apt-get install mysql-server` – verá que eles são parecidos. A principal diferença é a natureza **declarativa** da linguagem do Puppet. No comando manual, instruímos o gerenciador de pacotes a instalar o pacote `mysql-server` enquanto que no Puppet declaramos o estado final desejado do sistema.

Manifestos não são uma lista de comandos a ser executados. Quando o Puppet executa, ele compara o estado atual do sistema com o estado declarado

no manifesto, calcula a diferença e executa apenas as mudanças necessárias. Após salvar o arquivo, execute o Puppet com o seguinte comando:

```
vagrant@db$ sudo puppet apply db.pp
err: /Stage[main]//Package[mysql-server]/ensure: change from
purged to present failed: Execution of ...
...
E: Unable to fetch some archives, maybe run apt-get update or ...

notice: Finished catalog run in 6.62 seconds
```

Você pode ver que o Puppet tentou alterar o estado do pacote `mysql-server` de removido (*purged*) para presente (*present*), no entanto encontrou erros pois esquecemos de executar o comando `apt-get update`, como fizemos no capítulo 2. Para isso, vamos declarar um novo recurso `exec` e definir uma dependência dizendo que ele deve rodar antes do pacote `mysql-server`. Mudaremos o conteúdo do arquivo `db.pp` para:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { "mysql-server":
  ensure  => installed,
  require => Exec["apt-update"],
}
```

Para entender o conteúdo do manifesto, precisamos conhecer mais algumas características da linguagem do Puppet. Todo recurso possui um nome – a string entre aspas duplas logo após a declaração e antes do ":" – que serve como um identificador único. Quando um recurso precisa se referir a outro recurso, usamos a sintaxe com a primeira letra maiúscula e o nome do recurso entre colchetes.

O Puppet não garante que a ordem de execução respeite a ordem em que os recursos são declarados no manifesto. Quando existe uma dependência, você precisa declará-la explicitamente. Neste caso, adicionamos o parâmetro `require` no recurso `package` para garantir que o comando `apt-get`

`update` execute antes da instalação do MySQL. Agora o Puppet instalará o pacote `mysql-server` quando executado novamente:

```
vagrant@db$ sudo puppet apply db.pp
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//Package[mysql-server]/ensure: ensure
        changed \
'purged' to 'present'
notice: Finished catalog run in 59.98 seconds
```

Você pode ver que o Puppet alterou o estado do pacote `mysql-server` de removido (*purged*) para presente (*present*). Para confirmar que o pacote foi realmente instalado, você pode executar o comando:

```
vagrant@db$ aptitude show mysql-server
Package: mysql-server
State: installed
...
```

Perceba que o gerenciador de pacotes diz que o estado atual do pacote `mysql-server` é "instalado" (*installed*). Para entender a natureza declarativa do manifesto, tente rodar o Puppet novamente:

```
vagrant@db$ sudo puppet apply db.pp
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: Finished catalog run in 6.41 seconds
```

Dessa vez, o recurso `apt-update` executou, porém nada aconteceu com o pacote `mysql-server`, que já estava instalado, satisfazendo o estado declarado em nosso manifesto do Puppet.

Outra diferença que você pode ter percebido entre o manifesto Puppet e o comando manual do capítulo 2 é que não precisamos dizer para o Puppet qual gerenciador de pacotes usar. Isso acontece pois recursos do Puppet são abstratos. Existem várias implementações diferentes – chamadas de **provedores** – para os diversos sistemas operacionais. Para o recurso `package`, o Puppet possui diversos provedores: `apt`, `rpm`, `yum`, `gem`, dentre outros. Na

hora da execução, com base em informações do sistema o Puppet irá escolher o provedor mais adequado. Para ver uma lista com todos os provedores disponíveis para um determinado recurso assim como documentação detalhada dos parâmetros que o recurso aceita, você pode executar o comando:

```
vagrant@db$ puppet describe package

package
=======
Manage packages. ...
...
Providers
---------
    aix, appdmg, apple, apt, aptitude, aptrpm, blastwave, dpkg,
    fink, freebsd, gem, hpux, macports, msi, nim, openbsd,
    pacman, pip, pkg, pkgdmg, pkgutil, portage, ports,
    portupgrade, rpm, rug, sun, sunfreeware, up2date, urpmi,
    yum, zypper
```

Agora que entendemos o funcionamento básico do Puppet, vamos parar de executar comandos manuais dentro do servidor e começar a escrever código de infraestrutura fora da máquina virtual. Usaremos o Vagrant para nos ajudar a provisionar o servidor.

Recomeçar usando Puppet com Vagrant

Primeiramente vamos sair da máquina virtual `db` e destruí-la novamente, como fizemos no fim do capítulo 3:

```
vagrant@db$ logout
Connection to 127.0.0.1 closed.
$ vagrant destroy db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Forcing shutdown of VM...
==> db: Destroying VM and associated drives...
```

No mesmo diretório em que você criou o arquivo `Vagrantfile`, crie um novo diretório chamado `manifests` e, dentro dele, um novo arquivo `db.pp`

com o mesmo conteúdo da seção anterior. A nova estrutura de diretórios deve estar assim:

```
.
 Vagrantfile
 manifests
     db.pp
```

Agora é preciso alterar o arquivo de configuração do Vagrant – o arquivo `Vagrantfile` – para usar o Puppet como ferramenta de provisionamento da máquina virtual `db`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
    db_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

Feito isso, assim que subirmos uma nova máquina virtual para o servidor de banco de dados, o Vagrant irá rodar o Puppet automaticamente, sem precisarmos logar no servidor por SSH e executar comandos manualmente.

A saída da execução do Puppet será mostrada logo após a saída normal do Vagrant e você perceberá se algum erro acontecer. Por exemplo, a execução inicial mostrará o pacote `mysql-server` sendo instalado:

```
$ vagrant up db
Bringing machine 'db' up with 'virtualbox' provider...
==> db: Importing base box 'hashicorp/precise32'...
==> db: Matching MAC address for NAT networking...
==> db: Checking if box 'hashicorp/precise32' is up to date...
==> db: Setting the name of the VM: blank_db_1394854957677_21160
==> db: Clearing any previously set network interfaces...
==> db: Preparing network interfaces based on configuration...
    db: Adapter 1: nat
    db: Adapter 2: hostonly
==> db: Forwarding ports...
    db: 22 => 2222 (adapter 1)
==> db: Running 'pre-boot' VM customizations...
==> db: Booting VM...
==> db: Waiting for machine to boot. This may take a few
        minutes...
    db: SSH address: 127.0.0.1:2222
    db: SSH username: vagrant
    db: SSH auth method: private key
==> db: Machine booted and ready!
==> db: Configuring and enabling network interfaces...
==> db: Mounting shared folders...
    db: /vagrant => /private/tmp/blank
    db: /tmp/vagrant-puppet-2/manifests => /private/tmp/blank/
                                                      manifests
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//Package[mysql-server]/ensure: ensure
        changed \'purged' to 'present'
notice: Finished catalog run in 71.68 seconds
```

Agora que sabemos como usar o Vagrant e o Puppet para provisionar

nosso servidor, é hora de restaurar o ambiente de produção da loja virtual.

4.4 Reinstalando o servidor de banco de dados

Temos um manifesto simples que executa o primeiro passo da instalação do servidor de banco de dados. Se você lembra do próximo passo que tomamos no capítulo 2, precisamos nesse momento criar o arquivo de configuração `/etc/mysql/conf.d/allow_external.cnf` para permitir acesso remoto ao servidor MySQL. Faremos isso usando o recurso `file`, que representa a existência de um arquivo no sistema. Alteramos o conteúdo do arquivo `db.pp` para:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { "mysql-server":
  ensure  => installed,
  require => Exec["apt-update"],
}

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => "[mysqld]\n  bind-address = 0.0.0.0",
  require => Package["mysql-server"],
}
```

Por padrão, o nome do recurso representa o caminho completo onde o arquivo será criado no sistema e o parâmetro `content` possui o conteúdo do arquivo. Para especificar um caminho diferente, você pode usar o parâmetro `path`. Os parâmetros `owner`, `group` e `mode` representam o usuário e grupo associados ao arquivo e suas permissões.

Para aplicar a nova configuração, ao invés de destruir e subir uma nova máquina virtual, você pode usar o comando `provision` do Vagrant para simplesmente executar o Puppet na máquina virtual já existente:

```
$ vagrant provision db
  ==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//File[ /etc/mysql/conf.d/
                                          allow_external.cnf]/ \
ensure: defined content as
                  '{md5}4149205484cca052a3bfddc8ae60a71e'
notice: Finished catalog run in 4.36 seconds
```

Dessa vez, o Puppet criou o arquivo `/etc/mysql/conf.d/allow_external.cnf`. Para saber quando um arquivo de configuração precisa ser alterado, ele calcula um *checksum* MD5 do seu conteúdo. O MD5 é um algoritmo de *hash* bastante usado para verificar a integridade de um arquivo. Uma pequena alteração no conteúdo do arquivo faz com que o seu *checksum* MD5 seja totalmente diferente.

No nosso caso, o conteúdo do arquivo de configuração é pequeno e conseguimos declará-lo por completo no manifesto. No entanto, é comum ter arquivos de configuração com dezenas ou centenas de linhas. Gerenciar arquivos longos como uma única string em código Puppet não é uma boa ideia. Podemos trocar o conteúdo do arquivo por um ***template***. O Puppet entende *templates* ERB (*embedded ruby*), um sistema de *template* que permite embutir código Ruby dentro de um arquivo texto.

Para usar um *template*, primeiramente precisamos extrair o conteúdo para um novo arquivo `allow_ext.cnf`, que deve ficar junto do manifesto:

```
.
 Vagrantfile
 manifests
     allow_ext.cnf
     db.pp
```

Mude o conteúdo do arquivo `allow_ext.cnf` ligeiramente, para que o Puppet altere o arquivo na próxima execução:

```
[mysqld]
   bind-address = 9.9.9.9
```

Por fim, trocamos o parâmetro `content` do recurso `file` para usar o *template*:

```
exec ...
package ...

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => template("/vagrant/manifests/allow_ext.cnf"),
  require => Package["mysql-server"],
}
```

Ao executar o Puppet novamente, você verá que o *checksum* MD5 irá mudar e o conteúdo do arquivo será alterado:

```
$ vagrant provision db
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//File[ /etc/mysql/conf.d/
                                          allow_external.cnf]/ \
content: content changed
  '{md5}4149205484cca052a3bfddc8ae60a71e'
  \ to
  '{md5}8e2381895fcf40fa7692e38ab86c7192'
notice: Finished catalog run in 4.60 seconds
```

Agora precisamos reiniciar o serviço para que o MySQL perceba a alteração no arquivo de configuração. Fazemos isso declarando um novo recurso do tipo `service`:

```
exec ...
package ...
file ...

service { "mysql":
```

```
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["mysql-server"],
}
```

O serviço possui novos parâmetros: `ensure => running`, que garante que o serviço esteja rodando; `enable`, que garante que o serviço rode sempre que o servidor reiniciar; `hasstatus` e `hasrestart`, que declaram que o serviço entende os comandos `status` e `restart`; por fim, o parâmetro `require`, que declara uma dependência com o recurso `Package["mysql-server"]`.

Se você tentar executar o Puppet, verá que nada acontece pois o serviço já foi iniciado quando o pacote foi instalado. Para reiniciar o serviço toda vez que o arquivo de configuração mudar, precisamos declarar uma nova dependência no recurso `File["/etc/mysql/conf.d/allow_external.cnf"]`:

```
exec ...
package ...

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => template("/vagrant/manifests/allow_ext.cnf"),
  require => Package["mysql-server"],
  notify  => Service["mysql"],
}

service ...
```

O parâmetro `notify` define uma dependência de execução: sempre que o recurso `file` for alterado, ele fará com que o recurso `service` execute. Dessa vez, se alterarmos o conteúdo do *template* `allow_ext.cnf` de volta para `0.0.0.0` e executarmos o Puppet, o serviço será reiniciado:

```
$ vagrant provision db
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//File[ /etc/mysql/conf.d/
                                            allow_external.cnf]/ \
content: content changed
'{md5}8e2381895fcf40fa7692e38ab86c7192' \
to
'{md5}907c91cba5e1c7770fd430182e42c437'
notice: /Stage[main]//Service[mysql]: Triggered 'refresh' \
from 1 events
notice: Finished catalog run in 6.88 seconds
```

Com essa alteração, o banco de dados está rodando novamente e uma das verificações do Nagios deve ficar verde. Para corrigir a próxima verificação precisamos completar a configuração do banco de dados criando o *schema* e o usuário da loja virtual.

Criaremos o *schema* declarando mais um recurso `exec` com o mesmo comando usado no capítulo 2:

```
exec ...
package ...
file ...
service ...

exec { "loja-schema":
 unless  => "mysql -uroot loja_schema",
 command => "mysqladmin -uroot create loja_schema",
  path    => "/usr/bin/",
 require => Service["mysql"],
}
```

Além da dependência com o recurso `Service["mysql"]`, usamos um novo parâmetro `unless` que especifica um comando de teste: caso seu código de saída seja zero, o comando principal não irá executar. Essa é a forma de tornar um recurso do tipo `exec` idempotente. Ao contrário do `exec`

que usamos para rodar o `apt-get update`, neste caso é importante que o Puppet não tente criar um novo *schema* toda vez que for executado.

Conforme fizemos no capítulo 2, precisamos remover a conta anônima de acesso ao MySQL antes de criar o usuário para acessar nosso novo *schema*. Faremos isso declarando um novo recurso `exec`:

```
exec ...
package ...
file ...
service ...
exec ...

exec { "remove-anonymous-user":
  command => "mysql -uroot -e \"DELETE FROM mysql.user \
                              WHERE user=''; \
                              FLUSH PRIVILEGES\"",
  onlyif  => "mysql -u' '",
  path    => "/usr/bin",
  require => Service["mysql"],
}
```

Neste comando, ao invés de usar o parâmetro `unless` usamos o parâmetro oposto `onlyif` que irá executar o comando principal apenas se o código de saída do comando de teste for zero. Este comando de teste tenta acessar o MySQL usando um usuário vazio. Caso consiga conectar, o Puppet precisa executar o comando SQL `DELETE ...` para remover a conta anônima. Por fim, criaremos um último recurso `exec` para criar o usuário com permissão de acesso ao *schema* da loja virtual:

```
exec ...
package ...
file ...
service ...
exec ...
exec ...

exec { "loja-user":
  unless  => "mysql -uloja -plojasecret loja_schema",
```

```
  command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                              loja_schema.* TO 'loja'@'%' \
                              IDENTIFIED BY 'lojasecret';\"",
  path    => "/usr/bin/",
  require => Exec["loja-schema"],
}
```

Ao executar o Puppet pela última vez no servidor de banco de dados, veremos uma saída parecida com:

```
$ vagrant provision db
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//Exec[loja-schema]/returns: executed
        successfully
notice: /Stage[main]//Exec[loja-user]/returns: executed
        successfully
notice: /Stage[main]//Exec[remove-anonymous-user]/returns:
        executed \ successfully
notice: Finished catalog run in 15.56 seconds
```

Com isso conseguimos restaurar por completo o servidor de banco de dados e todas as verificações do Nagios referentes ao *host* `db` deverão ficar verdes. A única verificação que permanecerá vermelha é a verificação do Tomcat, pois a aplicação ainda não está acessível. Nas próximas seções iremos automatizar o processo de provisionamento do servidor `web` e do *deploy* da aplicação. Por enquanto, o conteúdo completo do manifesto `db.pp` ficou:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { "mysql-server":
  ensure  => installed,
  require => Exec["apt-update"],
}
```

```
file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => template("/vagrant/manifests/allow_ext.cnf"),
  require => Package["mysql-server"],
  notify  => Service["mysql"],
}

service { "mysql":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["mysql-server"],
}

exec { "loja-schema":
  unless  => "mysql -uroot loja_schema",
  command => "mysqladmin -uroot create loja_schema",
  path    => "/usr/bin/",
  require => Service["mysql"],
}

exec { "remove-anonymous-user":
  command => "mysql -uroot -e \"DELETE FROM mysql.user \
                              WHERE user=''; \
                              FLUSH PRIVILEGES\"",
  onlyif  => "mysql -u' '",
  path    => "/usr/bin",
  require => Service["mysql"],
}

exec { "loja-user":
  unless  => "mysql -uloja -plojasecret loja_schema",
  command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                              loja_schema.* TO 'loja'@'%' \
                              IDENTIFIED BY 'lojasecret';\"",
```

```
  path    => "/usr/bin/",
  require => Exec["loja-schema"],
}
```

4.5 Reinstalando o servidor web

Agora que o servidor `db` está restaurado, vamos automatizar o processo de provisionamento e *deploy* no servidor `web`. Para isto, criaremos um novo arquivo no diretório `manifests` chamado `web.pp`:

```
.
 Vagrantfile
 manifests
     allow_ext.cnf
     db.pp
     web.pp
```

Em vez de associar o novo manifesto com o servidor `web` já existente, vamos criar uma nova máquina virtual temporária chamada `web2` para podermos testar nosso código Puppet isoladamente antes de aplicá-lo no ambiente de produção. Ao final do capítulo, iremos nos livrar da máquina virtual `web2` e aplicaremos a mesma configuração no servidor `web` de verdade. O conteúdo do `Vagrantfile` com a nova máquina virtual deve ficar:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
    db_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
```

```
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
  end

  config.vm.define :web2 do |web_config|
    web_config.vm.network :private_network,
                          :ip => "192.168.33.13"
    web_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "web.pp"
    end
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

Inicialmente vamos declarar apenas os recursos para executar o `apt-get update` e instalar os pacotes base do servidor `web2`, de forma similar aos comandos executados no capítulo 2:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { ["mysql-client", "tomcat7"]:
  ensure  => installed,
  require => Exec["apt-update"],
}
```

Ao subir a nova máquina virtual, o Puppet irá aplicar essa configuração e instalar os pacotes:

```
$ vagrant up web2
Bringing machine 'web2' up with 'virtualbox' provider...
==> web2: Importing base box 'hashicorp/precise32'...
...
==> web2: Running provisioner: puppet...
```

```
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//Package[mysql-client]/ensure: ensure
        changed \ 'purged' to 'present'
notice: /Stage[main]//Package[tomcat7]/ensure: ensure changed \
        'purged' to 'present'
notice: Finished catalog run in 105.89 seconds
```

Feito isso, você pode abrir seu navegador e digitar a URL http://192.168.33.13:8080/ e você deverá ver uma página que diz *"It Works!"*. Isso significa que o Tomcat está instalado e rodando corretamente.

Os próximos passos realizados no capítulo 2 foram a configuração da conexão segura com HTTPS, a criação do `keystore` para armazenar o certificado SSL e o aumento da quantidade de memória disponível para a máquina virtual Java quando o Tomcat inicia.

Todas essas operações envolvem editar ou criar um novo arquivo no sistema. Já sabemos como entregar arquivos com o Puppet, então faremos os três passos de uma só vez. Primeiramente, vamos reaproveitar os arquivos de configuração que já estão funcionando no servidor `web` como *templates* para o manifesto do Puppet. Os seguintes comandos irão copiar os arquivos da máquina virtual `web` para o diretório compartilhado pelo Vagrant:

```
$ vagrant ssh web -- 'sudo cp /var/lib/tomcat7/conf/.keystore \
> /vagrant/manifests/'
$ vagrant ssh web -- 'sudo cp /var/lib/tomcat7/conf/server.xml \
> /vagrant/manifests/'
$ vagrant ssh web -- 'sudo cp /etc/default/tomcat7 \
> /vagrant/manifests/'
```

Em vez de usar o comando `vagrant ssh` para logar na máquina virtual, passamos um comando entre aspas após os traços `--`. Esse comando executará dentro da máquina virtual. O caminho `/vagrant` é onde o Vagrant mapeia o diretório onde você definiu o `Vagrantfile` dentro da máquina virtual. Se tudo der certo, os arquivos fora do Vagrant devem ser:

```
.
 Vagrantfile
 manifests
     .keystore
     allow_ext.cnf
     db.pp
     server.xml
     tomcat7
     web.pp
```

Como não precisamos alterar nada no conteúdo desses arquivos, basta declararmos novos recursos do tipo `file` para que o Puppet entregue-os no lugar certo, ajustando seus donos e permissões:

```
exec ...
package ...

file { "/var/lib/tomcat7/conf/.keystore":
  owner   => root,
  group   => tomcat7,
  mode    => 0640,
  source  => "/vagrant/manifests/.keystore",
  require => Package["tomcat7"],
}

file { "/var/lib/tomcat7/conf/server.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/server.xml",
  require => Package["tomcat7"],
}

file { "/etc/default/tomcat7":
  owner   => root,
  group   => root,
  mode    => 0644,
  source  => "/vagrant/manifests/tomcat7",
  require => Package["tomcat7"],
```

```
}
```

Por último, precisamos declarar o serviço `tomcat7`, que precisa ser reiniciado quando os arquivos de configuração são alterados. Os parâmetros do recurso `service` são os mesmos que usamos no serviço `mysql` quando configuramos o servidor de banco de dados:

```
exec ...
package ...

file { "/var/lib/tomcat7/conf/.keystore":
  ...
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/conf/server.xml":
  ...
  notify  => Service["tomcat7"],
}

file { "/etc/default/tomcat7":
  ...
  notify  => Service["tomcat7"],
}

service { "tomcat7":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["tomcat7"],
}
```

Podemos então rodar o manifesto todo para aplicar as últimas configurações no servidor `web2`:

```
$ vagrant provision web2
==> web2: Running provisioner: puppet...
Running Puppet with web.pp...
```

```
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//File[ /var/lib/tomcat7/conf/server.xml]/
        content: \ content changed
        '{md5}523967040584a921450af2265902568d' \
        to '{md5}e4f0d5610575a720ad19fae4875e9f2f'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf/.keystore]/
        ensure: \ defined content as
        '{md5}80792b4dc1164d67e1eb859c170b73d6'
notice: /Stage[main]//File[ /etc/default/tomcat7]/content:
        content \
        changed '{md5}49f3fe5de425aca649e2b69f4495abd2' to \
        '{md5}1f429f1c7bdccd3461aa86330b133f51'
notice: /Stage[main]//Service[tomcat7]: Triggered 'refresh' from
        3 events
notice: Finished catalog run in 14.58 seconds
```

Feito isso, temos o Tomcat configurado para aceitar conexões HTTP e HTTPS. Podemos testar abrindo uma nova janela no navegador web e acessando a URL https://192.168.33.13:8443/ . Após aceitar o certificado autoassinado, você verá uma página que diz *"It works!"*.

4.6 Fazendo deploy da aplicação

Para termos um servidor `web` completo, precisamos fazer o *deploy* da loja virtual. Em vez de usar o Puppet para reconstruir o processo de *build* dentro da máquina virtual – como fizemos no capítulo 2 – vamos reaproveitar o artefato `.war` gerado. No capítulo 6 iremos revisitar essa decisão e encontraremos uma forma melhor de gerar o artefato `.war` da aplicação.

Vamos também reaproveitar o *template* do arquivo `context.xml`, no qual definimos os recursos JNDI para acesso ao banco de dados. Usando o mesmo truque da seção anterior, vamos copiar os arquivos de dentro da máquina virtual `web`:

```
$ vagrant ssh web -- 'sudo cp /var/lib/tomcat7/conf/
                                                   context.xml \
```

```
> /vagrant/manifests/'
$ vagrant ssh web -- \
> 'sudo cp /var/lib/tomcat7/webapps/devopsnapratica.war \
> /vagrant/manifests/'
```

Agora o conteúdo dos arquivos e diretórios fora do Vagrant devem ser:

```
.
 Vagrantfile
 manifests
     .keystore
     allow_ext.cnf
     context.xml
     db.pp
     devopsnapratica.war
     server.xml
     tomcat7
     web.pp
```

O artefato contento a aplicação web `devopsnapratica.war` não precisa ser alterado, porém vamos definir variáveis dentro do manifesto para conter os dados de acesso ao banco de dados. Essas variáveis serão substituídas quando o Puppet processar o *template* ERB que criaremos no arquivo `context.xml`:

```
exec ...
package ...
file ...
file ...
file ...
service ...

$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
$db_password = "lojasecret"

file { "/var/lib/tomcat7/conf/context.xml":
  owner    => root,
```

```
  group   => tomcat7,
  mode    => 0644,
  content => template("/vagrant/manifests/context.xml"),
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war":
  owner   => tomcat7,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/devopsnapratica.war",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}
```

A sintaxe do Puppet para declarar variáveis é um cifrão `$` seguido do nome da variável. Usamos o símbolo de igual `=` para atribuir valores às nossas variáveis. Para usar essas variáveis dentro do *template* do arquivo `context.xml`, precisamos alterar o seu conteúdo substituindo os valores existentes pelo marcador `<%= var %>`, onde `var` deve ser o nome da variável sem o cifrão `$`. O novo *template* ERB do arquivo `manifests/context.xml` deve ser:

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>

    <Resource name="jdbc/web" auth="Container"
      type="javax.sql.DataSource" maxActive="100"
      maxIdle="30"
      maxWait="10000"
      username="<%= db_user %>"
      password="<%= db_password %>"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://<%= db_host %>:3306/<%= db_schema %>"/>

    <Resource name="jdbc/secure" auth="Container"
```

```
    type="javax.sql.DataSource" maxActive="100" maxIdle="30"
    maxWait="10000"
    username="<%= db_user %>"
    password="<%= db_password %>"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://<%= db_host %>:3306/<%= db_schema %>"/>

  <Resource name="jdbc/storage" auth="Container"
    type="javax.sql.DataSource" maxActive="100" maxIdle="30"
    maxWait="10000"
    username="<%= db_user %>"
    password="<%= db_password %>"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://<%= db_host %>:3306/<%= db_schema %>"/>
</Context>
```

Com isso, finalizamos o manifesto Puppet para provisionar e fazer *deploy* da aplicação no servidor `web2`. Podemos executá-lo rodando o comando:

```
$ vagrant provision web2
==> web2: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//File[ /var/lib/tomcat7/conf/context.xml]/
        content: \ content changed
        '{md5}4861cda2bbf3a56fbfdb78622c550019' \
        to '{md5}8365f14d15ddf99b1d20f9672f0b98c7'
notice: /Stage[main]//File[ /var/lib/tomcat7/webapps/ \
devopsnapratica.war]/ensure: defined content as \
'{md5}cba179ec04e75ce87140274b0aaa2263'
notice: /Stage[main]//Service[tomcat7]: Triggered 'refresh' from
        2 events
notice: Finished catalog run in 21.78 seconds
```

Tente agora acessar a URL http://192.168.33.13:8080/devopsnapratica/ no seu navegador para ver que a loja está funcionando corretamente. Sucesso!

Agora que sabemos recriar o servidor `web` por completo, podemos nos

livrar da máquina virtual `web2`. Para demonstrar o poder do uso de automação para provisionamento e *deploy*, vamos também destruir a máquina virtual `web` e recriá-la do zero:

```
$ vagrant destroy web web2
    web2: Are you sure you want to destroy the 'web2' VM?[y/N] y
==> web2: Forcing shutdown of VM...
==> web2: Destroying VM and associated drives...
==> web2: Running cleanup tasks for 'puppet' provisioner...
    web: Are you sure you want to destroy the 'web' VM? [y/N] y
==> web: Forcing shutdown of VM...
==> web: Destroying VM and associated drives...
```

Antes de reiniciar a máquina virtual novamente, precisamos atualizar o arquivo de configuração do Vagrant – o `Vagrantfile` – para conter apenas um servidor `web`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
    db_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
    web_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "web.pp"
    end
  end

  config.vm.define :monitor do |monitor_config|
```

```
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

O último passo é reconstruir a máquina virtual `web` do zero:

```
$ vagrant up web
Bringing machine 'web' up with 'virtualbox' provider...
==> web: Importing base box 'hashicorp/precise32'...
==> web: Matching MAC address for NAT networking...
==> web: Checking if box 'hashicorp/precise32' is up to date...
==> web: Setting the name of the VM:
                                    blank_web_1394856878524_4521
==> web: Fixed port collision for 22 => 2222. Now on port 2200.
==> web: Clearing any previously set network interfaces...
==> web: Preparing network interfaces based on configuration...
    web: Adapter 1: nat
    web: Adapter 2: hostonly
==> web: Forwarding ports...
    web: 22 => 2200 (adapter 1)
==> web: Running 'pre-boot' VM customizations...
==> web: Booting VM...
==> web: Waiting for machine to boot. This may take a few
         minutes...
    web: SSH address: 127.0.0.1:2200
    web: SSH username: vagrant
    web: SSH auth method: private key
==> web: Machine booted and ready!
==> web: Configuring and enabling network interfaces...
==> web: Mounting shared folders...
    web: /vagrant => /private/tmp/blank
    web: /tmp/vagrant-puppet-2/manifests =>
         /private/tmp/blank/manifests
==> web: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//Package[mysql-client]/ensure: ensure
```

```
        changed \ 'purged' to 'present'
notice: /Stage[main]//Package[tomcat7]/ensure: ensure changed
        'purged' \ to 'present'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf/context.xml]/
        content: \ content changed
        '{md5}4861cda2bbf3a56fbfdb78622c550019' \
        to '{md5}8365f14d15ddf99b1d20f9672f0b98c7'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf/server.xml]/
        content: \ content changed
        '{md5}523967040584a921450af2265902568d' \
        to '{md5}e4f0d5610575a720ad19fae4875e9f2f'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf/.keystore]/
        ensure: \ defined content as
        '{md5}80792b4dc1164d67e1eb859c170b73d6'
notice: /Stage[main]//File[ /var/lib/tomcat7/webapps/ \
        devopsnapratica.war]/ensure: defined content as \
        '{md5}cba179ec04e75ce87140274b0aaa2263'
notice: /Stage[main]//File[ /etc/default/tomcat7]/content:
        content \ changed
        '{md5}49f3fe5de425aca649e2b69f4495abd2' to \
        '{md5}1f429f1c7bdccd3461aa86330b133f51'
notice: /Stage[main]//Service[tomcat7]: Triggered 'refresh' from
        5 events
notice: Finished catalog run in 106.31 seconds
```

Pronto! Conseguimos reconstruir o servidor `web` e fazer *deploy* da loja virtual **em 1 minuto**!

A verificação do Nagios deve ficar verde e a loja virtual está novamente acessível para nossos usuários. A versão final do manifesto `web.pp`, ao final deste capítulo, ficou:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { ["mysql-client", "tomcat7"]:
  ensure  => installed,
  require => Exec["apt-update"],
}
```

```
file { "/var/lib/tomcat7/conf/.keystore":
  owner   => root,
  group   => tomcat7,
  mode    => 0640,
  source  => "/vagrant/manifests/.keystore",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/conf/server.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/server.xml",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/etc/default/tomcat7":
  owner   => root,
  group   => root,
  mode    => 0644,
  source  => "/vagrant/manifests/tomcat7",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

service { "tomcat7":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["tomcat7"],
}

$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
```

```
$db_password = "lojasecret"

file { "/var/lib/tomcat7/conf/context.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  content => template("/vagrant/manifests/context.xml"),
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war":
  owner   => tomcat7,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/devopsnapratica.war",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}
```

Investir em automação nos permitiu destruir e reconstruir servidores em questão de minutos. Você também deve ter percebido que passamos muito pouco tempo executando comandos manuais dentro dos servidores. Transferimos a responsabilidade de execução de comandos para o Puppet e declaramos o estado final desejado em arquivos de manifesto.

O código do Puppet está começando a crescer e não temos uma estrutura clara para organizar os arquivos de manifesto e os *templates*. Também estamos tratando o artefato `.war` como um arquivo estático, algo que sabemos não ser ideal. Nos próximos capítulos iremos revisitar essas decisões e criar um ecossistema mais robusto para entrega contínua da loja virtual.

Capítulo 5

Puppet além do básico

No final do capítulo 4, o código Puppet para configurar a infraestrutura da loja virtual não está muito bem organizado. A única separação que fizemos foi criar dois arquivos de manifesto, um para cada servidor: `web` e `db`. No entanto, dentro de cada arquivo, o código é simplesmente uma lista de recursos. Além disso, os arquivos de configuração e de *templates* estão jogados sem nenhuma estrutura. Neste capítulo vamos aprender novos conceitos e funcionalidades do Puppet ao mesmo tempo que refatoramos o código para torná-lo mais idiomático e bem organizado.

5.1 Classes e tipos definidos

O Puppet gerencia uma única instância de cada recurso definido em um manifesto, tornando-os uma espécie de *singleton*. Da mesma forma, uma **classe** é uma coleção de recursos únicos em seu sistema. Se você conhece lingua-

gens orientadas a objetos, não se confunda com a terminologia. Uma classe no Puppet não pode ser instanciada diversas vezes. Classes são apenas uma forma de dar nome a uma coleção de recursos que serão aplicados como uma unidade.

Um bom uso de classes no Puppet é para configuração de serviços que você precisa instalar apenas uma vez no sistema. Por exemplo, no arquivo `db.pp` instalamos e configuramos o servidor MySQL e criamos o *schema* e o usuário específicos da loja virtual. Na vida real, podemos ter vários *schemas* e usuários usando o mesmo banco de dados, porém não instalamos mais de um MySQL por sistema. O servidor MySQL é um bom candidato para definirmos em uma classe genérica, que chamaremos de `mysql-server`. Refatorando nosso arquivo `db.pp` para declarar e usar a nova classe, teremos:

```
class mysql-server {
  exec    { "apt-update": ... }
  package { "mysql-server": ... }
  file    { "/etc/mysql/conf.d/allow_external.cnf": ... }
  service { "mysql": ... }
  exec    { "remove-anonymous-user": ... }
}

include mysql-server

exec { "loja-schema": ...,
  require => Class["mysql-server"],
}
exec { "loja-user": ... }
```

Perceba que movemos o recurso `Exec["remove-anonymous-user"]` – que remove a conta de acesso anônima – para dentro da classe `mysql-server` pois isso é algo que deve acontecer somente quando o servidor MySQL é instalado pela primeira vez. Outra mudança que você pode perceber é que o recurso `Exec["loja-schema"]` agora tem uma dependência com `Class["mysql-server"]` em vez de um recurso específico `Service["mysql"]` como anteriormente. Essa é uma forma de encapsular detalhes de implementação dentro da classe, isolando esse conhecimento do resto do código, que pode declarar dependências em coisas mais abstratas e

estáveis.

Para definir uma nova classe basta escolher seu nome e colocar todos os recursos dentro de uma declaração do tipo `class <nome da classe> { ... }`. Para usar uma classe você pode utilizar a sintaxe `include <nome da classe>` ou a versão mais parecida com a definição de um recurso com que já estamos acostumados: `class { "<nome da classe>": ... }`.

Por outro lado, os recursos que criam o *schema* e o usuário da loja virtual podem ser reaproveitados para criar *schemas* e usuários para outras aplicações rodando neste mesmo servidor. Colocá-los em uma classe seria a forma errada de encapsulamento pois o Puppet exigiria que ela fosse única. Para situações como esta, o Puppet possui uma outra forma de encapsulamento chamada de "tipos definidos".

Um **tipo definido** – ou *defined type* – é uma coleção de recursos que pode ser usada várias vezes em um mesmo manifesto. Eles permitem que você elimine duplicação de código agrupando recursos relacionados que podem ser reutilizados em conjunto. Você pode interpretá-los como equivalentes a macros em uma linguagem de programação. Além disso, eles podem ser parametrizados e definir valores padrão para parâmetros opcionais. Agrupando os dois recursos `exec` para criar o *schema* e o usuário de acesso ao banco de dados em um tipo definido chamado `mysql-db`, teremos:

```
class mysql-server { ... }

define mysql-db($schema, $user = $title, $password) {
  Class['mysql-server'] -> Mysql-db[$title]

  exec { "$title-schema":
    unless  => "mysql -uroot $schema",
    command => "mysqladmin -uroot create $schema",
    path    => "/usr/bin/",
  }

  exec { "$title-user":
    unless  => "mysql -u$user -p$password $schema",
    command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                                $schema.* TO '$user'@'%' \
                                IDENTIFIED BY '$password';\"",
```

```
    path     => "/usr/bin/",
    require => Exec["$title-schema"],
  }
}

include mysql-server

mysql-db { "loja":
  schema   => "loja_schema",
  password => "lojasecret",
}
```

A sintaxe para declarar um tipo definido é `define <nome do tipo>(<parâmetros>) { ... }`. Nesse exemplo, o tipo definido `mysql-db` aceita três parâmetros: `$schema`, `$user` e `$password`. O parâmetro `$user`, quando não especificado, terá o mesmo valor do parâmetro especial `$title`. Para entender o valor do parâmetro `$title` – que não precisa ser declarado explicitamente – basta olhar a sintaxe da chamada na qual declaramos uma instância do tipo definido: `mysql-db { "loja": schema => "loja_schema", ... }`. O nome do recurso que instancia o tipo definido, neste caso `loja`, será passado como valor do parâmetro `$title`. Os outros parâmetros são passados seguindo a mesma sintaxe que usamos em outros recursos nativos do Puppet: o nome do parâmetro e seu valor separador por uma flecha `=>`.

Movemos os dois recursos `exec` que criam o *schema* e o usuário para dentro do tipo definido. Substituímos também todas as referências *hard coded* pelo respectivos parâmetros, tomando cuidado para usar aspas duplas nas strings para que o Puppet saiba expandir os valores corretamente. Para poder usar o tipo definido mais de uma vez, precisamos parametrizar o nome dos recursos `exec` e torná-los únicos usando o parâmetro `$title`. A última modificação foi promover a declaração da dependência com a classe `mysql-server` para o topo do tipo definido. Com isso, os comandos individuais dentro do tipo definido não precisam declarar dependências com recursos externos, facilitando a manutenção desse código no futuro.

Usando classes e tipos definidos, conseguimos refatorar o código Puppet para torná-lo um pouco mais reutilizável. No entanto, não mudamos a or-

ganização dos arquivos em si. Tudo continua declarado dentro de um único arquivo. Precisamos aprender uma forma melhor de organizar nossos arquivos.

5.2 Empacotamento e distribuição usando módulos

O Puppet define um padrão para empacotar e estruturar seu código: **módulos**. Módulos possuem uma estrutura predefinida de diretórios onde você deve colocar seus arquivos, assim como alguns padrões de nomenclatura. Módulos também são uma forma de compartilhar código Puppet com a comunidade. O Puppet Forge (http://forge.puppetlabs.com/) é um site mantido pela Puppet Labs onde você pode encontrar diversos módulos escritos pela comunidade ou registrar um módulo que você escreveu e quer compartilhar.

Dependendo da sua experiência com diferentes linguagens, os equivalentes a um módulo Puppet nas comunidades Ruby, Java, Python e .NET seriam uma *gem*, um *jar*, um *egg* e uma *DLL*, respectivamente. A estrutura de diretórios simplificada para um módulo Puppet é:

```
<nome do módulo>/
 files
    ...
 manifests
    init.pp
 templates
    ...
 tests
     init.pp
```

Em primeiro lugar, o nome do diretório raiz define o nome do módulo. O diretório *manifests* é o mais importante pois lá você coloca seus arquivos de manifesto (com extensão `.pp`). Dentro dele deve existir pelo menos um arquivo chamado `init.pp`, que serve como ponto de entrada para o módulo. O diretório *files* contém arquivos de configuração estáticos que podem ser acessados dentro de um manifesto usando uma URL especial: `puppet:///modules/<nome do módulo>/<arquivo>`. O di-

retório *templates* contém arquivos ERB que podem ser referenciados dentro de um manifesto usando o nome do módulo: template('<nome do módulo>/<arquivo ERB>').

Por fim, o diretório *tests* contém exemplos mostrando como usar as classes e tipos definidos pelo módulo. Esses testes não fazem nenhum tipo de verificação automatizada. Você pode rodá-los usando o comando `puppet apply --noop`. Esse comando simula uma execução do Puppet sem realizar nenhuma alteração no sistema. Caso haja algum erro de sintaxe ou de compilação, você irá detectá-los na saída do comando `puppet apply --noop`.

Nossa próxima refatoração é uma preparação antes de criarmos o módulo. Vamos quebrar a definição da classe `mysql-server` e do tipo definido `mysql-db` em dois arquivos separados chamados `server.pp` e `db.pp`, respectivamente. No mesmo nível em que criamos o arquivo `Vagrantfile`, vamos criar um novo diretório `modules` onde colocaremos nossos módulos. Dentro dele, criamos um novo módulo para instalação e configuração do MySQL, chamado `mysql`, criando a seguinte estrutura de diretórios e movendo os respectivos arquivos para lá:

```
.
 Vagrantfile
 manifests
    .keystore
    context.xml
    db.pp
    devopsnapratica.war
    server.xml
    tomcat7
    web.pp
 modules
     mysql
         manifests
            init.pp
            db.pp
            server.pp
         templates
            allow_ext.cnf
```

Perceba que criamos um novo arquivo chamado `init.pp` no diretório

`modules/mysql/manifests`. Esse arquivo será o ponto de entrada para o módulo e ele deve declarar uma classe `mysql` vazia que serve como *namespace* para o resto do módulo:

```
class mysql { }
```

Movemos também o arquivo `allow_ext.cnf` para o diretório `modules/mysql/templates`. Por estar dentro da estrutura padrão do módulo, não precisamos mais referenciar o *template* usando um caminho absoluto. O Puppet aceita o caminho no formato `<nome do módulo>/<arquivo>` e sabe procurar o *template* `<arquivo>` dentro do módulo escolhido. Podemos então mudar o recurso `file` no arquivo `modules/mysql/manifests/server.pp` de `template("/vagrant/manifests/allow_ext.cnf")` para `template("mysql/allow_ext.cnf")`:

```
class mysql::server {
  exec { "apt-update": ... }
  package { "mysql-server": ... }

  file { "/etc/mysql/conf.d/allow_external.cnf":
    ...
    content => template("mysql/allow_ext.cnf"),
    ...
  }

  service { "mysql": ... }
  exec { "remove-anonymous-user": ... }
}
```

Perceba que renomeamos a classe `mysql-server` para `mysql::server`. Esta é a nomenclatura padrão que o Puppet entende para importar classes e tipos dentro do mesmo módulo. Da mesma forma, devemos também renomear o tipo definido `mysql-db` para `mysql::db` dentro do arquivo `modules/mysql/manifests/db.pp`, e alterar sua dependência com a classe `mysql::server`:

```
define mysql::db($schema, $user = $title, $password) {
  Class['mysql::server'] -> Mysql::Db[$title]
```

```
  ...
}
```

Por fim podemos simplificar bastante o arquivo `manifests/db.pp` para usar a classe `mysql::server` e o tipo definido `mysql::db` no novo módulo:

```
include mysql::server

mysql::db { "loja":
  schema   => "loja_schema",
  password => "lojasecret",
}
```

Com isso, criamos um módulo genérico que disponibiliza uma classe para instalar o servidor MySQL e um tipo definido para criar um *schema* e um usuário para acessá-lo. Pode parecer confuso o fato de estarmos mudando as coisas de lugar sem nenhuma mudança no comportamento, porém é exatamente por isso que refatorarmos código! Melhoramos sua estrutura sem modificar o comportamento externo. O benefício é facilitar o trabalho de manutenção no futuro, isolando componentes individuais e agrupando aquilo que é relacionado. Princípios de *design* de software – como encapsulamento, acoplamento e coesão – também ajudam a melhorar código de infraestrutura.

Para testar que tudo continua funcionando, você pode destruir e subir a máquina virtual `db` novamente, porém antes disso é preciso fazer uma pequena alteração para dizer ao Vagrant que temos um novo módulo que precisa estar disponível quando o Puppet executar. Mudamos o arquivo `Vagrantfile` para adicionar a configuração `module_path` nas VMs `web` e `db`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...
  config.vm.define :db do |db_config|
    ...
    db_config.vm.provision "puppet" do |puppet|
      puppet.module_path = "modules"
```

```
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
    ...
    web_config.vm.provision "puppet" do |puppet|
      puppet.module_path = "modules"
      puppet.manifest_file = "web.pp"
    end
  end

  ...
end
```

Agora que o código Puppet para gerenciar a infraestrutura do banco de dados está refatorado, é hora de melhorar o código que instala, configura e faz *deploy* da aplicação no servidor web.

5.3 Refatorando o código Puppet do servidor web

Agora que sabemos como refatorar nosso código Puppet, vamos criar um novo módulo chamado `tomcat` que será usado pelo servidor web. Em primeiro lugar, vamos simplesmente criar a estrutura de diretórios do novo módulo e mover os arquivos do diretório `manifests` para os seguintes diretórios dentro de `modules/tomcat`:

```
.
 Vagrantfile
 manifests
    .keystore
    db.pp
    devopsnapratica.war
    web.pp
 modules
     mysql
        ...
```

```
    tomcat
        files
           server.xml
           tomcat7
        manifests
        templates
            context.xml
```

Os únicos arquivos restantes no diretório `manifests` são as definições dos servidores (`db.pp` e `web.pp`), o arquivo WAR contento a aplicação da loja virtual e o arquivo `.keystore` contendo o certificado SSL. Fizemos isso para seguir um princípio importante no desenvolvimento de software: **separação de responsabilidades**. Se olharmos o conteúdo do arquivo `web.pp`, estamos misturando recursos mais genéricos – que instalam e configuram o Tomcat – com recursos específicos da loja virtual. Se simplesmente movermos tudo para o mesmo módulo, ele não será reutilizável. Pense sob o ponto de vista externo de uma outra equipe na sua empresa ou a comunidade em geral: um módulo que instala e configura o Tomcat é muito mais útil do que um módulo que configura o Tomcat com toda a loja virtual instalada.

Tendo isso em mente, o objetivo do resto da refatoração será extrair um módulo do Tomcat que não possua nenhuma referência a coisas específicas da loja virtual. Conforme formos movendo definições de recurso do arquivo `web.pp` para o módulo `tomcat`, vamos manter coisas específicas da loja virtual no manifesto do servidor e tornar os recursos do módulo `tomcat` mais genéricos.

Antes de começarmos a atacar os recursos do Tomcat, vamos começar com algo um pouco mais simples: extrair a parte que instala o cliente do MySQL para uma nova classe no módulo `mysql`. Criamos um novo arquivo `client.pp` dentro de `modules/mysql/manifests`, com o conteúdo:

```
class mysql::client {
  exec { "apt-update":
    command => "/usr/bin/apt-get update"
  }

  package { "mysql-client":
    ensure  => installed,
```

```
    require => Exec["apt-update"],
  }
}
```

Novamente, a motivação é separação de responsabilidades: a classe `mysql::client` pode ser útil em outros contextos, mesmo quando você não está desenvolvendo uma aplicação web, portanto colocá-la no módulo `mysql` faz mais sentido do que no módulo `tomcat`. Aos poucos estamos começando a desembaraçar o código.

Com isso podemos começar a mudar o arquivo `web.pp`: vamos incluir a nova classe `mysql::client`, remover o recurso `Exec[apt-update]` – que migrou para a nova classe – e precisamos também atualizar todas as referências aos arquivos estáticos e de *template* para o novo caminho dentro do módulo:

```
include mysql::client

package { "tomcat7": ... }

file { "/var/lib/tomcat7/conf/.keystore": ... }

file { "/var/lib/tomcat7/conf/server.xml": ...
  source  => "puppet:///modules/tomcat/server.xml", ...
}

file { "/etc/default/tomcat7": ...
  source  => "puppet:///modules/tomcat/tomcat7", ...
}

service { "tomcat7": ... }

$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
$db_password = "lojasecret"

file { "/var/lib/tomcat7/conf/context.xml": ...
  content => template("tomcat/context.xml"), ...
```

```
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

As únicas referências que devem continuar apontando para o caminho absoluto são os recursos `File[/var/lib/tomcat7/conf/.keystore]` e `File[/var/lib/tomcat7/webapps/devopsnapratica.war]`, já que não os movemos para dentro do módulo.

Agora vamos começar a mover partes do código do arquivo `web.pp` para novas classes e tipos definidos dentro do módulo `tomcat`. Criaremos um novo arquivo `server.pp` dentro do diretório `modules/tomcat/manifests` e iremos mover os seguintes recursos do arquivo `web.pp` para uma nova classe que chamaremos de `tomcat::sever`:

```
class tomcat::server {
  package { "tomcat7": ... }
  file { "/etc/default/tomcat7": ... }
  service { "tomcat7": ... }
}
```

Para expor esta nova classe no módulo novo, precisamos criar também um arquivo `init.pp` dentro do mesmo diretório, contendo inicialmente uma única classe vazia: `class tomcat { }`. Com isso, podemos alterar o arquivo `web.pp`, removendo os recursos que foram movidos e substituindo-os por um `include` da nova classe `tomcat::server`:

```
include mysql::client
include tomcat::server

file { "/var/lib/tomcat7/conf/.keystore": ... }
file { "/var/lib/tomcat7/conf/server.xml": ... }

$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
$db_password = "lojasecret"
```

```
file { "/var/lib/tomcat7/conf/context.xml": ... }
file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

Perceba que ainda não movemos todos os recursos do tipo `file` para a nova classe, pois eles possuem referências e configurações que são bastante específicas da nossa aplicação:

- O arquivo `.keystore` possui o certificado SSL da loja virtual. Não é uma boa ideia distribuir nosso certificado em um módulo genérico do Tomcat que outras pessoas ou equipes podem reutilizar.
- O arquivo `server.xml` possui configurações para habilitar HTTPS contendo informações secretas que também são específicas da loja virtual.
- O arquivo `context.xml` configura os recursos JNDI de acesso ao *schema* do banco de dados da loja virtual.
- Por fim, o arquivo `devopsnapratica.war` contém nossa aplicação.

Para tornar o módulo do Tomcat mais genérico, vamos primeiro resolver o problema da configuração SSL e do arquivo `.keystore`. Dentro do arquivo de configuração `server.xml`, em vez de assumir que o nome do *keystore* e a senha de acesso serão sempre os mesmos, vamos transformá-lo em um *template* e passar essas informações para a classe `tomcat::server`. Para entender o resultado desejado, começaremos mostrando as alterações no arquivo `web.pp`:

```
include mysql::client

$keystore_file = "/etc/ssl/.keystore"
$ssl_connector = {
  "port"          => 8443,
  "protocol"      => "HTTP/1.1",
  "SSLEnabled"    => true,
  "maxThreads"    => 150,
  "scheme"        => "https",
  "secure"        => "true",
```

```
  "keystoreFile" => $keystore_file,
  "keystorePass" => "secret",
  "clientAuth"   => false,
  "sslProtocol"  => "SSLv3",
}
$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
$db_password = "lojasecret"

file { $keystore_file:
  mode    => 0644,
  source  => "/vagrant/manifests/.keystore",
}

class { "tomcat::server":
  connectors => [$ssl_connector],
  require    => File[$keystore_file],
}

file { "/var/lib/tomcat7/conf/context.xml": ... }
file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

Vamos discutir cada uma das mudanças: em primeiro lugar, mudamos o diretório do arquivo `.keystore` para um lugar mais genérico, fora do Tomcat, e salvamos sua localização em uma nova variável `$keystore_file`. Com isso, removemos as referências e dependências com o Tomcat do recurso `File[/etc/ssl/.keystore]`. A segunda mudança importante é que tornamos a classe `tomcat::server` parametrizada, passando um novo parâmetro `connectors` que representa um *array* de conectores que precisam ser configurados.

Esse exemplo também serve para mostrar duas novas estruturas de dados reconhecidas pelo Puppet: *arrays* e *hashes*. Um *array* é uma lista indexada de itens, enquanto um *hash* é uma espécie de dicionário que associa uma chave a um valor. Ao passar a lista de conectores como um parâmetro, podemos mover o recurso `File[/var/lib/tomcat7/conf/server.xml]` para dentro da classe `tomcat::server`, no arquivo

modules/tomcat/manifests/server.pp:

```
class tomcat::server($connectors = []) {
  package { "tomcat7": ... }
  file { "/etc/default/tomcat7": ... }

  file { "/var/lib/tomcat7/conf/server.xml":
    owner   => root,
    group   => tomcat7,
    mode    => 0644,
    content => template("tomcat/server.xml"),
    require => Package["tomcat7"],
    notify  => Service["tomcat7"],
  }

  service { "tomcat7": ... }
}
```

Perceba que a sintaxe de declaração da classe mudou para aceitar o novo parâmetro `$connectors`, que possui um valor padrão correspondente a uma lista vazia. Como agora o conteúdo do arquivo `server.xml` precisa ser dinâmico, precisamos movê-lo do diretório `modules/tomcat/files` para o diretório `modules/tomcat/templates`, substituindo também a definição do recurso para usar o *template* na geração do arquivo. Por fim, precisamos mudar o conteúdo do arquivo `server.xml` para configurar os conectores dinamicamente:

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
  ...
  <Service name="Catalina">
    ...
    <Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               URIEncoding="UTF-8"
               redirectPort="8443" />

    <%- connectors.each do |c| -%>
      <Connector
```

```
        <%= c.sort.map{|k,v| "#{k}='#{v}'"}.join(" ") %> />
    <%- end -%>
    ...
  </Service>
</Server>
```

Perceba que agora nenhum conector referencia qualquer característica específica da nossa aplicação. Usando um pouco de Ruby no *template* `server.xml` conseguimos transformar em dados o que antes estava *hard-coded*. A parte interessante é a linha: `c.sort.map{|k,v| "#{k}='#{v}'"}.join(" ")`. A operação `sort` garante que sempre iremos iterar no *hash* na mesma ordem. A operação `map` transforma cada par de chave/valor do *hash* em uma *string* formatada como um atributo XML. Por exemplo: um hash `{"port"=> "80", "scheme"=> "http"}` se transformará na lista `["port='80'", "scheme='http'"]`. A operação `join` junta todas as *strings* dessa lista em uma única *string*, separando-as com um espaço. O resultado final é a *string*: `"port='80'scheme='http'"`.

Agora que resolvemos o problema da configuração SSL, podemos usar o mesmo truque para mover o recurso `File[/var/lib/tomcat7/conf/context.xml]` para dentro da classe `tomcat::server`. Novamente começamos pela mudança desejada no arquivo `web.pp`:

```
include mysql::client

$keystore_file = "/etc/ssl/.keystore"
$ssl_conector  = ...
$db = {
  "user"     => "loja",
  "password" => "lojasecret",
  "driver"   => "com.mysql.jdbc.Driver",
  "url"      => "jdbc:mysql://192.168.33.10:3306/loja_schema",
}

file { $keystore_file: ... }

class { "tomcat::server":
```

```
  connectors   => [$ssl_conector],
  data_sources => {
    "jdbc/web"     => $db,
    "jdbc/secure"  => $db,
    "jdbc/storage" => $db,
  },
  require      => File[$keystore_file],
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

Seguindo o mesmo padrão, adicionamos um novo parâmetro `data_sources` na classe `tomcat::server` e passamos um *hash* contendo os dados de configuração de cada *data source*. Movemos também o recurso `File[/var/lib/tomcat7/conf/context.xml]` para a classe `tomcat::server` no arquivo `modules/tomcat/manifests/server.pp`:

```
class tomcat::server($connectors = [], $data_sources = []) {
  package { "tomcat7": ... }
  file { "/etc/default/tomcat7": ... }
  file { "/var/lib/tomcat7/conf/server.xml": ... }

  file { "/var/lib/tomcat7/conf/context.xml":
    owner   => root,
    group   => tomcat7,
    mode    => 0644,
    content => template("tomcat/context.xml"),
    require => Package["tomcat7"],
    notify  => Service["tomcat7"],
  }

  service { "tomcat7": ... }
}
```

Precisamos também alterar o arquivo de *template* `context.xml` para gerar os recursos JNDI dinamicamente:

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
```

```
    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>

    <%- data_sources.sort.each do |data_source, db| -%>
      <Resource name="<%= data_source %>" auth="Container"
            type="javax.sql.DataSource" maxActive="100"
            maxIdle="30"
            maxWait="10000"
            username="<%= db['user'] %>"
            password="<%= db['password'] %>"
            driverClassName="<%= db['driver'] %>"
            url="<%= db['url'] %>"/>
    <%- end -%>
</Context>
```

Com isso conseguimos refatorar bastante nosso código, tornado-os mais organizado e mais fácil de ser reaproveitado. Nesse momento, a estrutura de diretórios com os novos módulos deve ser:

```
.
 Vagrantfile
 manifests
    .keystore
    db.pp
    devopsnapratica.war
    web.pp
 modules
     mysql
        manifests
           client.pp
           db.pp
           init.pp
           server.pp
        templates
            allow_ext.cnf
     tomcat
         files
            tomcat7
         manifests
```

```
        init.pp
        server.pp
    templates
        context.xml
        server.xml
```

5.4 Separação de responsabilidades: infraestrutura vs. aplicação

Até agora, estamos nos forçando a escrever módulos que sejam genéricos e reaproveitáveis. No entanto, o código que restou nos manifestos `web.pp`, `db.pp` e os arquivos soltos `.keytore` e `devopsnapratica.war` ainda parecem um mau cheiro. Uma prática comum para resolver esse problema é criar módulos específicos para a aplicação. Mas isso não contradiz os argumentos que viemos fazendo até agora?

Separar módulos de infraestrutura dos módulos de aplicação é uma forma de aplicar o princípio da separação de responsabilidades em outro nível. Essa é uma forma de composição: os módulos de aplicação ainda utilizam os módulos de infraestrutura, que continuam reaproveitáveis e independentes. A figura 5.1 mostra a estrutura e dependências desejadas entre nossos módulos.
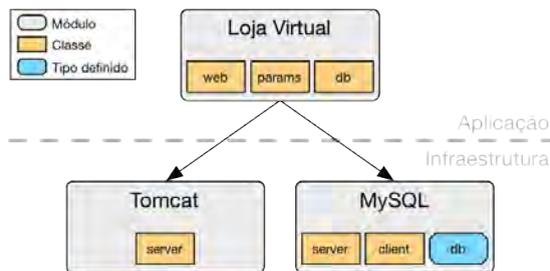
Figura 5.1: Módulos de aplicação vs. módulos de infraestrutura

O último módulo que iremos criar neste capítulo será responsável pela instalação e configuração da loja virtual. Para isso, precisamos expandir nossa estrutura de diretórios e arquivos da mesma forma que fizemos anteriormente, mas dessa vez para o novo módulo `loja_virtual`:

```
.
 Vagrantfile
 manifests
    db.pp
    web.pp
 modules
     loja_virtual
        files
           .keystore
           devopsnapratica.war
        manifests
            db.pp
            init.pp
            params.pp
            web.pp
     mysql
        ...
     tomcat
         ...
```

Movemos os arquivos .keystore e devopsnapratica.war para o diretório modules/loja_virtual/files e criamos quatro arquivos de manifesto para colocarmos as novas classes do módulo loja_virtual: db.pp, init.pp, params.pp e web.pp. O arquivo init.pp simplesmente declara uma classe vazia para servir de *namespace* raiz para o resto do módulo:

```
class loja_virtual { }
```

O arquivo db.pp irá definir uma nova classe loja_virtual::db, com o conteúdo movido quase sem alterações do arquivo manifest/db.pp:

```
class loja_virtual::db {
  include mysql::server
  include loja_virtual::params

  mysql::db { $loja_virtual::params::db['user']:
    schema   => $loja_virtual::params::db['schema'],
    password => $loja_virtual::params::db['password'],
```

```
  }
}
```

A única diferença é que movemos os valores do usuário, senha e *schema* do banco de dados para uma outra classe `loja_virtual::params` dentro do mesmo módulo. Este é um padrão comum em módulos Puppet, em que você centraliza os parâmetros em uma única classe para não precisar duplicá-los dentro do módulo.

Os parâmetros definidos na classe `loja_virtual::params` também respeitam seu *namespace*, por isso precisamos referenciá-los usando o caminho completo `$loja_virtual::params::db['user']` em vez de simplesmente `$db['user']`. Para entender melhor como esses parâmetros foram definidos, basta olhar o conteúdo do arquivo `params.pp` que define a nova classe `loja_virtual::params`:

```
class loja_virtual::params {
  $keystore_file = "/etc/ssl/.keystore"

  $ssl_connector = {
    "port"          => 8443,
    "protocol"      => "HTTP/1.1",
    "SSLEnabled"    => true,
    "maxThreads"    => 150,
    "scheme"        => "https",
    "secure"        => "true",
    "keystoreFile"  => $keystore_file,
    "keystorePass"  => "secret",
    "clientAuth"    => false,
    "sslProtocol"   => "SSLv3",
  }

  $db = {
    "user"     => "loja",
    "password" => "lojasecret",
    "schema"   => "loja_schema",
    "driver"   => "com.mysql.jdbc.Driver",
    "url"      => "jdbc:mysql://192.168.33.10:3306/",
  }
```

```
}
```

Esse é praticamente o mesmo conteúdo que tínhamos no arquivo `manifests/web.pp`. A única diferença foi a adição da chave `schema` ao *hash* `$db`, cujo valor estava anteriormente embutido no final do parâmetro `$db['url']`. Fizemos isto para poder reaproveitar a mesma estrutura nas classes `loja_virtual::db` e `loja_virtual::web`. Com esta alteração, você precisa também mudar o arquivo de template `modules/tomcat/templates/context.xml` para usar a nova chave `schema`:

```
...
<Context>
  ...
  <Resource ... url="<%= db['url'] + db['schema'] %>"/>
  ...
</Context>
```

Como a definição dos parâmetros já está disponível em uma única classe, fica fácil de entender o código movido e alterado na nova classe `loja_virtual::web`:

```
class loja_virtual::web {
  include mysql::client
  include loja_virtual::params

  file { $loja_virtual::params::keystore_file:
    mode    => 0644,
    source  => "puppet:///modules/loja_virtual/.keystore",
  }

  class { "tomcat::server":
    connectors => [$loja_virtual::params::ssl_connector],
    data_sources => {
      "jdbc/web"     => $loja_virtual::params::db,
      "jdbc/secure"  => $loja_virtual::params::db,
      "jdbc/storage" => $loja_virtual::params::db,
    },
    require => File[$loja_virtual::params::keystore_file],
```

```
  }

  file { "/var/lib/tomcat7/webapps/devopsnapratica.war":
    owner   => tomcat7,
    group   => tomcat7,
    mode    => 0644,
    source  => "puppet:///modules/loja_virtual/
                devopsnapratica.war",
    require => Package["tomcat7"],
    notify  => Service["tomcat7"],
  }
}
```

Da mesma forma que fizemos na classe `loja_virtual::db`, incluímos a classe `loja_virtual::params` e usamos o caminho completo para os parâmetros. Também mudamos os recursos do tipo `file` para referenciar os arquivos usando o caminho relativo ao módulo ao invés do caminho absoluto no sistema de arquivos.

Com isso, nosso novo módulo está completo e o conteúdo dos arquivos de manifesto de cada servidor fica bastante simplificado:

```
# Conteúdo do arquivo manifests/web.pp:
include loja_virtual::web

# Conteúdo do arquivo manifests/db.pp:
include loja_virtual::db
```

5.5 Puppet forge: reutilizando módulos da comunidade

Conforme o desenvolvimento de uma aplicação vai progredindo e o sistema cresce, é comum também aumentar o número de classes, pacotes e bibliotecas usadas no projeto. Com isso ganhamos uma nova preocupação: **gerenciamento de dependências**. Esse é um problema tão comum que diversas comunidades de desenvolvimento criaram ferramentas específicas para lidar com ele: na comunidade Ruby, o Bundler gerencia dependências entre *gems*; na comunidade Java, o Maven e o Ivy gerenciam *jars* e suas dependências; até

mesmo administradores de sistema usam gerenciadores de pacotes como o dpkg ou o rpm para cuidar das dependências na instalação de software.

O código de automação de infraestrutura da loja virtual também está começando a mostrar problemas no gerenciamento de suas dependências. Já temos três módulos Puppet e criamos uma dependência indesejada entre recursos nos módulos mysql e tomcat. Se você prestar atenção, o recurso Exec[apt-update] é declarado duas vezes nas classes mysql::server e mysql::client e é usado em três lugares diferentes: nas próprias classes e na classe tomcat::server.

De modo geral, dependências são mais simples de gerenciar quando a declaração e o uso estão mais próximos. Uma dependência entre recursos declarados no mesmo arquivo é melhor que uma dependência entre recursos declarados em arquivos diferentes; uma dependência entre arquivos diferentes do mesmo módulo é melhor que uma dependência com recursos de módulos diferentes.

Do jeito que o código está, o módulo tomcat tem uma dependência direta com um recurso específico do módulo mysql. Se a loja virtual deixar de usar o módulo mysql, não será possível instalar e configurar o Tomcat. Esse tipo de dependência cria um acoplamento forte entre os módulos mysql e tomcat, tornando mais difícil utilizá-los independentemente.

Idealmente, não precisaríamos declarar nenhum recurso específico do APT. O único motivo de termos declarado esse recurso no capítulo 4 foi porque precisávamos atualizar o índice do APT – executando um apt-get update – antes de instalar qualquer pacote. Isso não é uma necessidade do MySQL ou do Tomcat, mas sim um problema que aparece assim que o Puppet tenta instalar o primeiro pacote no sistema.

Para melhorar nosso código e remover essa dependência indesejada, vamos aproveitar para aprender como usar módulos da comunidade. Vamos usar um módulo da PuppetLabs que foi escrito para lidar com o APT de forma mais geral.

Ao ler a documentação do módulo (https://github.com/puppetlabs/puppetlabs-apt) vemos que ele expõe a classe apt e que ela possui um parâmetro always_apt_update que faz o que precisamos:

```
class { 'apt':
```

```
  always_apt_update => true,
}
```

Com isso, garantimos que o Puppet irá rodar o comando `apt-get update` toda vez que for executado. No entanto, precisamos garantir que o comando rode antes da instalação de qualquer pacote no sistema. Para declarar essa dependência, vamos aprender uma nova sintaxe do Puppet:

```
Class['apt'] -> Package <| |>
```

A flecha `->` impõe uma restrição na ordem de execução dos recursos. Da mesma forma que o parâmetro `require` indica uma dependência entre dois recursos, a flecha `->` garante que o recurso do lado esquerdo – nesse caso a classe `apt` – seja executado antes do recurso do lado direito. A sintaxe `<| |>` – também conhecida como operador espaçonave – é um **coletor de recursos**. O coletor representa um grupo de recursos e é composto de: um tipo de recurso, o operador `<|`, uma expressão de busca opcional e o operador `|>`. Como deixamos a expressão de busca em branco, selecionamos todos os recursos do tipo `package`.

Juntando essas duas informações temos um código mais genérico que garante que o Puppet irá executar um `apt-get update` antes de tentar instalar qualquer pacote no sistema. Com isso, podemos alterar a nossa classe `loja_virtual`, declarada no arquivo `modules/loja_virtual/manifests/init.pp`:

```
class loja_virtual {
  class { 'apt':
    always_apt_update => true,
  }

  Class['apt'] -> Package <| |>
}
```

Podemos agora remover as referências ao recurso `Exec[apt-update]` dos módulos `mysql` e `tomcat`, assim como o recurso em si. O arquivo `modules/mysql/manifests/client.pp` fica bem mais simplificado:

```
class mysql::client {
  package { "mysql-client":
    ensure  => installed,
  }
}
```

Da mesma forma, o arquivo `modules/mysql/manifests/server.pp` também fica mais enxuto:

```
class mysql::server {
  package { "mysql-server":
    ensure  => installed,
  }

  file { "/etc/mysql/conf.d/allow_external.cnf": ... }
  service { "mysql": ... }
  exec { "remove-anonymous-user": ... }
}
```

Por fim, removemos a dependência indesejada na classe `tomcat::server` alterando o arquivo `modules/tomcat/manifests/server.pp`:

```
class tomcat::server($connectors = [], $data_sources = []) {
  package { "tomcat7":
    ensure  => installed,
  }

  file { "/etc/default/tomcat7": ... }
  file { "/var/lib/tomcat7/conf/server.xml": ... }
  file { "/var/lib/tomcat7/conf/context.xml": ... }
  service { "tomcat7": ... }
}
```

Para completar essa refatoração, precisamos garantir que o código que adicionamos à classe `loja_virtual` seja executado nos servidores `web` e `db`. Para isso basta incluí-la no começo da declaração das classes `loja_virtual::web` e `loja_virtual::db`:

```
# Conteúdo do arquivo modules/loja_virtual/manifests/web.pp:
class loja_virtual::web {
  include loja_virtual
  ...
}

# Conteúdo do arquivo modules/loja_virtual/manifests/db.pp:
class loja_virtual::db {
  include loja_virtual
  ...
}
```

Pronto, nossa refatoração está completa. Refatorações alteram a estrutura interna do código sem alterar seu comportamento externo [10]. Para garantir que nossos servidores ainda são configurados corretamente depois de todas as alterações feitas neste capítulo, vamos reprovisioná-los do zero, primeiro destruindo o servidor `db`:

```
$ vagrant destroy db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Forcing shutdown of VM...
==> db: Destroying VM and associated drives...
==> db: Running cleanup tasks for 'puppet' provisioner...
```

Porém, ao tentar subir o servidor `db` novamente, encontramos um problema inesperado:

```
$ vagrant up db
Bringing machine 'db' up with 'virtualbox' provider...
...
Puppet::Parser::AST::Resource failed with error ArgumentError:
Could not find declared class apt at /tmp/vagrant-puppet-2/
modules-0/loja_virtual/manifests/init.pp:4 on node precise32
```

Esse erro indica que o Puppet não conseguiu encontrar a classe `apt`, pois não instalamos o módulo da PuppetLabs. A forma mais simples de instalar um módulo da comunidade é usando a própria ferramenta de linha de comando do Puppet, executando: `puppet module install <nome-do-módulo>`. Isso irá instalar o novo módulo e suas dependências

no `modulepath` padrão, tornando-os disponíveis na próxima execução de um `puppet apply`.

Na atual configuração, não rodamos `puppet apply` diretamente. Estamos usamos o Vagrant para gerenciar a execução do Puppet. Além disso, o módulo precisa ser instalado dentro da máquina virtual, o que torna o processo de provisionamento um pouco mais complicado se quisermos executar um comando antes do Vagrant iniciar a execução do Puppet.

Conforme o número de módulos e suas dependências aumentam, fica mais difícil instalá-los um por um usando a linha de comando. A biblioteca **Librarian Puppet** (http://librarian-puppet.com/) foi criada justamente para resolver esse problema. Você declara todas as dependências em um único arquivo `Puppetfile` e o Librarian Puppet resolve e instala os módulos nas versões especificadas em um diretório isolado. Ele funciona de forma similar ao Bundler, para quem conhece o ecossistema Ruby.

No mesmo nível em que está o arquivo `Vagrantfile`, vamos criar um novo diretório `librarian` contendo o arquivo `Puppetfile` com o seguinte conteúdo:

```
forge "http://forge.puppetlabs.com"

mod "puppetlabs/apt", "1.4.0"
```

A primeira linha declara que usaremos o PuppetLabs Forge como fonte oficial para resolver e baixar os módulos. A segunda linha declara uma dependência com o módulo `pupppetlabs/apt` na versão `1.4.0`.

Para executar o Librarian Puppet antes de usar o Puppet, vamos instalar um novo *plugin* do Vagrant. *Plugins* são uma forma de estender o comportamento básico do Vagrant, permitindo que a comunidade escreva e compartilhe diversas melhorias. Outros exemplos de *plugins* do Vagrant são: suporte a outras ferramentas de virtualização, máquinas virtuais na nuvem, etc. Para instalar o *plugin* do Librarian Puppet (https://github.com/mhahn/vagrant-librarian-puppet/) , basta executar:

```
$ vagrant plugin install vagrant-librarian-puppet
Installing the 'vagrant-librarian-puppet' plugin. This can
  take a few minutes...
Installed the plugin 'vagrant-librarian-puppet (0.6.0)'!
```

Como estamos gerenciando os módulos do Librarian Puppet em um diretório isolado, precisamos configurá-lo no arquivo `Vagrantfile` e adicioná-lo no `modulepath`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...
  config.librarian_puppet.puppetfile_dir = "librarian"

  config.vm.define :db do |db_config|
    ...
    db_config.vm.provision "puppet" do |puppet|
      puppet.module_path = ["modules", "librarian/modules"]
      ...
    end
  end

  config.vm.define :web do |web_config|
    ...
    web_config.vm.provision "puppet" do |puppet|
      puppet.module_path = ["modules", "librarian/modules"]
      ...
    end
  end
  ...
end
```

Com isso, podemos tentar provisionar a máquina virtual `db` novamente e, dessa vez, se tudo der certo, veremos que após nosso código de automação de infraestrutura continua funcionando mesmo após todas as refatorações feitas até aqui:

```
$ vagrant reload db
==> db: Checking if box 'hashicorp/precise32' is up to date...
...
$ vagrant provision db
==> db: Installing Puppet modules with Librarian-Puppet...
==> db: Running provisioner: puppet...
```

```
Running Puppet with db.pp...
...
notice: Finished catalog run in 56.78 seconds
```

5.6 Conclusão

Missão cumprida! Comparado com o começo do capítulo, não apenas melhoramos a estrutura e a organização do nosso código como também aprendemos novas funcionalidades e características do Puppet. Os arquivos que estavam grandes e confusos terminaram o capítulo com apenas uma linha de código cada. Agora os arquivos de manifesto representam o que será instalado em cada servidor. Aprendemos também como reutilizar módulos da comunidade e como gerenciar dependências utilizando ferramentas de código livre como o Librarian Puppet e o respectivo *plugin* do Vagrant.

Como um exercício para reforçar seu entendimento, imagine se quiséssemos instalar a loja virtual completa num único servidor: quão difícil seria esta tarefa agora que temos módulos bem definidos? Um outro exercício interessante é usar o que aprendemos sobre o Puppet para automatizar a instalação e configuração do servidor de monitoramento do capítulo 3.

Mas é claro que ainda há espaço para melhoria. Escrever código limpo e de fácil manutenção é mais difícil que simplesmente escrever código que funciona. Precisamos refletir e decidir quando o código está bom o suficiente para seguir em frente.

Por enquanto, seguindo o espírito de melhoria contínua, decidimos parar de refatorar mesmo sabendo que ainda existem maus cheiros no nosso código. Em particular, entregar um arquivo WAR estático como parte de um módulo Puppet só funciona se você nunca precisar alterá-lo. Sabemos que isso não é verdade na maioria das situações, então iremos revisitar essa decisão nos próximos capítulos. Mas antes disso precisamos aprender uma forma segura e confiável de gerar um novo arquivo WAR toda vez que fizermos uma alteração no código da loja virtual.

Capítulo 6

Integração contínua

Se você é um desenvolvedor de software, até agora deve ter aprendido um pouco sobre o mundo de operações: subimos o ambiente de produção, fizemos *build* e *deploy* da aplicação, configuramos um sistema de monitoramento e escrevemos código Puppet para automatizar a infraestrutura. Deixamos de lado o que acontece antes da aplicação estar empacotada em um arquivo WAR para focar no lado operacional quando tentamos colocar um sistema em produção.

A essência de DevOps é melhorar a colaboração entre equipes de desenvolvimento e operações então é hora de olhar um pouco mais para o lado do desenvolvimento. Escrever código é uma das principais atividades de um desenvolvedor de software e DevOps pode ajudar a tornar o processo de entrega mais confiável. Vamos discutir como lidar com mudanças no código-fonte de forma segura, levando-as desde um *commit* até um pacote que pode ser implantado em produção.

6.1 Práticas de engenharia ágil

Muitas empresas começam sua adoção de métodos ágeis usando Scrum [14] ou Kanban [2], processos que focam nas práticas de gestão como: reuniões em pé, quadro de tarefas, divisão do trabalho em histórias, planejamento em iterações, ou retrospectivas. No entanto, praticantes de métodos ágeis experientes sabem que uma adoção de sucesso não pode deixar de lado as práticas de engenharia ágil que afetam diretamente a qualidade do software desenvolvido.

A programação extrema – *Extreme Programming* ou XP – foi uma das primeiras metodologias ágeis que revolucionou a forma como software era desenvolvido no final da década de 90. No livro seminal *Extreme Programming Explained* [4], Kent Beck definiu os valores, princípios e práticas da metodologia, cujo principal objetivo é permitir o desenvolvimento de software de alta qualidade que se adapta a requisitos vagos ou em constante mudança.

Dentre as práticas de engenharia introduzidas por XP estão: refatoração, TDD, propriedade coletiva de código, *design* incremental, programação em pares, padrões de código e integração contínua. Enquanto algumas práticas ainda geram controvérsia, muitas delas já podem ser consideradas normais hoje em dia, sendo adotadas por um grande número de empresas e equipes. Em particular, a prática da integração contínua foi uma das mais bem sucedidas por ajudar a resolver um grande problema dos processos de desenvolvimento de software existentes até então: a fase de integração tardia.

Nós já discutimos no capítulo 1 o que acontece quando deixamos para integrar e testar somente na "última milha". Por isso é importante dedicar um capítulo de um livro de DevOps à prática da integração contínua. Porém, antes de definirmos o que ela é e como implementá-la, precisamos falar sobre alguns de seus elementos básicos para entender porque ela funciona.

6.2 Começando pelo básico: controle de versões

Independente de você trabalhar isoladamente ou em equipe, o primeiro sinal de um programador bem disciplinado é a forma como ele gerencia mudanças no código-fonte do seu sistema. Quando eu estava aprendendo a programar,

escrevia código descartável e simplesmente mantinha uma cópia de todos os arquivos no meu computador. Isso funcionou até a primeira vez que precisei reverter o código para a última versão estável. Eu tinha feito tantas mudanças de uma só vez que o código ficou numa versão inconsistente, que não compilava e não funcionava. Tudo que eu queria era ter uma máquina do tempo que me levasse para a última versão que funcionava.

Quando comecei a trabalhar em equipe, o problema piorou. Eu e meus colegas queríamos fazer mudanças em partes diferentes do sistema, mas como cada um tinha uma cópia completa do código, precisávamos compartilhar diretórios, mandar e-mail com arquivos anexados, ou usar disquetes (sim, naquela época isso era comum) para transferir arquivos para lá e para cá. Não preciso dizer que essa era uma tarefa árdua e cometíamos erros esporádicos que tomavam um bom tempo para descobrir o que faltou ou como resolver conflitos.

Minha vida mudou quando descobri que esse problema já tinha sido resolvido por pessoas mais inteligentes e experientes que eu. Elas criaram os **sistemas de controle de versões**, ferramentas que atuam como um repositório central de código e mantêm um histórico de todas as mudanças feitas no seu sistema. Cada mudança é chamada de um *commit*, que agrupa os arquivos alterados com o autor da mudança e uma mensagem explicativa. Isso permite que você navegue no histórico do projeto e saiba quem mudou o que, quando, e por quê. Quando duas pessoas tentam alterar o mesmo arquivo, essas ferramentas tentam resolver o conflito automaticamente, porém lhe avisam se precisarem de ajuda humana.

Hoje em dia não existe motivo para não usar um sistema de controle de versões. Uma das ferramentas mais populares é o **Git** (http://git-scm.com/) , um sistema distribuído que não impõe a necessidade de existir um único repositório central para todos os *commits* do projeto. Por ser distribuído, cada desenvolvedor pode rodar um servidor local em sua própria máquina. O que o torna poderoso é que cada servidor sabe enviar seu histórico local para outros servidores, permitindo que você compartilhe código de forma eficiente. A escolha de um servidor central é apenas uma convenção, mas ninguém precisa estar conectado na rede para fazer seus *commits*.

É incrível como ainda encontro equipes que não têm disciplina alguma no

seu processo de entrega. Desenvolvedores simplesmente logam no servidor de produção e alteram o código "ao vivo", sem nenhuma forma de auditoria ou controle de qualidade. Qualquer desenvolvedor pode criar um defeito em produção e ninguém conseguirá saber a origem do problema.

Entrega contínua e DevOps exigem disciplina e colaboração. O processo de entrega de uma equipe disciplinada **sempre** começa com um *commit*. Qualquer mudança que precisa ir para produção – seja código-fonte, código de infraestrutura ou configuração – se inicia com um *commit* no sistema de controle de versões. Isso permite rastrear possíveis problemas em produção com sua origem, seu autor, além de servir como uma ferramenta central de colaboração entre as equipes de desenvolvimento e de operações.

A loja virtual está usando o Git como sistema de controle de versões. Seu repositório principal está hospedado no Github. Além de oferecer repositórios Git gratuitos para projetos de código aberto, o Github é uma ferramenta de colaboração que permite outros usuários relatarem defeitos, pedidos de novas funcionalidades, ou até mesmo clonarem repositórios para contribuir código que corrige ou implementa tais funcionalidades.

Em qualquer repositório Git, você pode usar o comando `git log` para mostrar o histórico de *commits* do projeto, por exemplo:

```
$ git log
commit 337865fa21a30fb8b36337cac98021647e03ddc2
Author: Danilo Sato <dtsato@gmail.com>
Date:   Tue Mar 11 14:31:22 2014 -0300

    Adicionando LICENSE

commit 64cff6b54816fe08f14fb3721c04fe7ee47d43c0
Author: Danilo Sato <dtsato@gmail.com>
...
```

Cada *commit* no Git é representado por um *hash* – ou SHA – que é um conjunto de letras e números que servem como identidade única. No exemplo anterior, o *commit* mais recente possui o SHA que começa com `337865da`. O SHA é calculado com base no conteúdo dos arquivos modificados pelo *commit* e ele funciona como uma espécie de chave primária que pode ser usada para referenciar o *commit*.

Em outros sistemas de controle de versão, como no SVN, *commits* são representados por números que autoincrementam a cada mudança, no entanto, como o Git é um sistema distribuído, cada repositório precisa calcular um *hash* que é globalmente único, independente da existência de outros repositórios.

Mais para frente veremos como criar novos *commits* no repositório local e como compartilhá-los com o repositório central. Por enquanto vamos aprender outras ferramentas e práticas de engenharia ágil que facilitam a adoção de DevOps e entrega contínua.

6.3 Automatizando o build do projeto

Já vimos o poder da automação nos capítulos anteriores quando substituímos um processo de provisionamento e *deploy* manual com código Puppet. A automação de tarefas repetitivas ajuda a diminuir o risco de erros humanos, além de encurtar o tempo de *feedback* entre a introdução de um problema e sua detecção. Automação é um dos principais pilares de DevOps e se mostra presente em diversas práticas de engenharia ágil.

Uma parte do processo de entrega que se beneficia bastante da automação é o **processo de *build***. O *build* de um sistema envolve todas as tarefas necessárias para conseguir executá-lo, como por exemplo: compilação, *download* e resolução de dependências, vinculação com bibliotecas, empacotamento, cálculo de métricas de qualidade etc. Essas tarefas podem variar dependendo da linguagem de programação que você usa. Em Ruby, por exemplo, você não precisa compilar arquivos antes de executá-los.

O resultado do *build* é geralmente um ou mais arquivos binários, também chamados de **artefatos**. Em um sistema Java, alguns exemplos de artefatos são: arquivos `.jar`, `.war` ou até mesmo documentação Javadoc em HTML gerada a partir do código-fonte ou relatórios gerados pela ferramenta que calcula métricas de qualidade do código.

Um exemplo de processo de *build* não relacionado ao desenvolvimento de software está sendo usado na escrita deste livro. O livro é escrito em arquivos texto usando uma linguagem de marcação desenvolvida pela Caelum – o **Tubaína** (https://github.com/caelum/tubaina) . Nosso processo de *build*

transforma os arquivos texto e imagens em artefatos que podem ser impressos ou lidos como e-books em formato PDF, `.mobi` ou `.epub`.

Conforme um sistema cresce, seu processo de *build* fica mais complicado e é difícil lembrar todos os passos necessários caso você esteja executando-os manualmente. Por isso é importante investir em automação. Atualmente, todas as principais linguagens possuem uma ou mais ferramentas de código livre disponíveis para ajudar no processo de *build*: em Java temos o Ant, o Maven, o Buildr, o Gradle; em Ruby temos o Rake; em Javascript temos o Grunt; em .NET temos o NAnt; além de diversas outras alternativas.

A loja virtual foi escrita usando o Maven como ferramenta de *build*. Conforme discutimos no final do capítulo 2, nosso processo de *build* irá resolver e baixar todas as dependências, compilar, executar testes automatizados e empacotar os artefatos de 4 módulos distintos: `core`, `admin`, `site` e `combined`. Para testar que o *build* do Maven ainda funciona no seu clone do repositório, você pode executar o comando:

```
$ mvn install
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------
...
[INFO] ------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------
[INFO] loja-virtual-devops ............... SUCCESS [1.156s]
[INFO] core .............................. SUCCESS [2.079s]
[INFO] admin ............................. SUCCESS [10:16.469s]
[INFO] site .............................. SUCCESS [7.928s]
[INFO] combined .......................... SUCCESS [25.543s]
[INFO] ------------------------------------------------------
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------
[INFO] Total time: 10 minutes 53 seconds
[INFO] Finished at: Thu Mar 14 07:15:24 UTC 2013
[INFO] Final Memory: 80M/179M
[INFO] ------------------------------------------------------
```

Por enquanto, não vamos fazer nenhuma alteração no processo de *build*,

porém um aspecto importante da sua execução merece um pouco mais de atenção: os testes automatizados.

6.4 Testes automatizados: diminuindo risco e aumentando a confiança

Compilar o sistema é um passo importante para sua execução, principalmente em linguagens de tipagem estática. No entanto, não é garantia que o sistema irá funcionar corretamente. Ele pode conter defeitos ou até mesmo ter funcionalidades implementadas de forma errada. Por isso é importante também incluir no processo de *build* a execução de **testes automatizados**.

Existem diversos tipos de testes que podem ser automatizados, cada um com características distintas. Não existe uma classificação ou terminologia amplamente aceita na indústria, porém vejamos alguns dos tipos de teste mais comuns:

- **Testes de unidade**: práticas como TDD [3] e refatoração [10] dão bastante ênfase nesse tipo de teste, que exercita unidades de código isoladamente sem a necessidade de subir e aplicação inteira. Esses testes geralmente rodam bem rápido – na ordem de milissegundos – podendo ser executados com bastante frequência e fornecendo o melhor tipo de *feedback* para os desenvolvedores.

- **Testes de integração** ou **testes de componente**: exercitam uma parte da aplicação ou como ela se integra com suas dependências, por exemplo: um banco de dados, um serviço REST, uma fila de mensagens, ou até mesmo os *frameworks* e bibliotecas utilizados. Dependendo do componente que está sendo testado, a execução desses testes podem exigir que partes do sistema estejam rodando.

- **Testes funcionais** ou **testes de aceitação**: exercitam as funcionalidades do sistema como um todo, do ponto de vista externo. Geralmente este tipo de teste exercita a interface gráfica ou simula a interação com o sistema do ponto de vista de um usuário. Esses testes geralmente são mais lentos pois exigem que a maior parte do sistema esteja rodando.

E esses não são os únicos tipos de testes, existem muito mais: testes de desempenho, testes exploratórios, testes de usabilidade etc. Eles contribuem para avaliar a qualidade do seu sistema, porém nem sempre é fácil automatizá-los. Testes exploratórios, por exemplo, são manuais por natureza e exigem a criatividade de um ser humano para descobrir cenários onde o sistema não se comporta da forma esperada.

A comunidade ágil geralmente se refere aos quatro quadrantes da figura 6.1 descritos por Brian Marick [7] como um modelo para descrever os tipos de teste necessários para entregar software de qualidade.
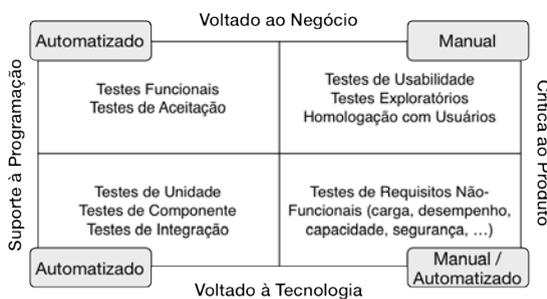
Figura 6.1: Quadrantes de teste

No lado esquerdo estão testes que dão suporte à programação, ou testes que os desenvolvedores executam com bastante frequência para obter *feedback* sobre as mudanças introduzidas em cada *commit*. Esses testes se preocupam em avaliar a qualidade interna do software. No lado direito estão testes que criticam o produto como um todo, avaliando sua qualidade externa e tentando encontrar problemas perceptíveis a um usuário do sistema.

No lado superior estão testes voltados ao negócio. Esses testes são descritos do ponto de vista do negócio, usando uma linguagem que faz sentido para um especialista de domínio. No lado inferior estão testes voltados à tecnologia. Esses testes são descritos do ponto de vista técnico, usando termos que farão mais sentido para os programadores mas não tanto para um especialista de domínio.

O diagrama mostra exemplos de tipos de teste para cada quadrante e também mostra onde é mais comum investir em automação. Testes no quadrante

superior direito são geralmente manuais, pois exigem criatividade ou exploração, tarefas que humanos são bons em executar. Testes no quadrante inferior esquerdo são geralmente automatizados para poder ser executados diversas vezes e evitar defeitos de regressão. Nessa diagonal, não existe muita controvérsia na indústria e muitas empresas adotam essa estratégia de testes. A outra diagonal é um pouco mais controversa.

Um dos grandes problemas na indústria de software é a forma como empresas encaram testes do quadrante superior esquerdo. É muito comum ver empresas abordarem esses testes de forma manual e investir uma enorme quantidade de tempo e pessoas escrevendo *scripts* de teste em uma planilha ou um sistema de gerenciamento de casos de teste.

Humanos são péssimos para executar esses *scripts* pass a passo pois não somos bons em tarefas repetitivas: é comum pularmos uma etapa ou até executá-la na ordem errada. Por esse motivo ainda existem empresas alocando uma quantidade considerável de tempo na "última milha" para uma fase de testes abrangente. Uma abordagem ágil para os testes do quadrante superior esquerdo é investir em automação para que os humanos tenham mais tempo para executar testes no quadrante superior direito onde podem adicionar muito mais valor.

No quadrante inferior direito o problema é um pouco diferente. Existem formas de automatizar alguns tipos de teste deste quadrante, enquanto outros ainda precisam ser manuais. No entanto, o problema mais comum é que esse quadrante é geralmente ignorado. Problemas de desempenho ou escalabilidade só são encontrados em produção, quando já é tarde demais. Esta é uma área onde a colaboração trazida por DevOps pode trazer grandes benefícios para a entrega de software de qualidade.

No caso da loja virtual, como grande parte do sistema é implementada pelo *Broadleaf Commerce*, temos pouco código Java customizado e, consequentemente, poucos testes automatizados. A loja de demonstração original não tinha nenhum teste, apesar de o código da biblioteca em si ser bem testado. Poderíamos escrever um outro livro apenas abordando esse assunto, porém decidimos automatizar apenas alguns testes de unidade e funcionais para usarmos como exemplos.

Nosso principal interesse é usar esses testes para fornecer *feedback* quando

introduzimos mudanças. Quanto mais tempo se passa entre o momento em que um defeito é introduzido até sua detecção, mais difícil será corrigi-lo. Por esse motivo é importante incluir a maior quantidade possível de testes automatizados no processo de *build*.

O Maven espera que você escreva testes de unidade e, inclusive, já sabe como rodá-los no processo de *build* padrão. Se você examinar a saída do comando `mvn install` que executamos na seção anterior, verá o resultado dos testes de unidade no *build* do componente `admin`:

```
...
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running br.com.devopsnapratica.....AdminLoginControllerTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time ...

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
...
```

O nosso *build* do Maven também está configurado para executar testes funcionais que escrevemos usando o **Selenium** (https://code.google.com/p/selenium/) , também conhecido atualmente como **WebDriver**. O Selenium é uma biblioteca para automação do navegador web, que nos permite simular o comportamento de um usuário navegando pela loja virtual e escrever testes automatizados no estilo JUnit para verificar que tudo funciona como o esperado.

No módulo `site` criamos apenas um exemplo de teste funcional definido na classe `br.com.devopsnapratica.acceptance.SearchTest`. Inicialmente precisamos escolher qual navegador iremos utilizar e qual URL ele deve acessar:

```
@Before
public void openBrowser() throws InterruptedException {
    driver = new HtmlUnitDriver();
    driver.get("http://localhost:8080/");
```

```
    Thread.sleep(5000); // Waiting for Solr index
}
```

A anotação `@Before` é parte do *JUnit* e significa que o método será executado antes de qualquer teste. Nesse caso, criamos uma instância de um `HtmlUnitDriver` que é a implementação mais rápida e leve de um `WebDriver` pois, ao invés de abrir um novo processo e uma nova janela para o navegador web, ele emula tudo em Java. Outras implementações existem para os navegadores mais comuns, como por exemplo: `ChromeDriver`, `FirefoxDriver` ou se estiver no Windows, `InternetExplorerDriver`.

O método `get(String url)` carrega uma página web na URL especificada, bloqueando até que a operação termine. Por fim, fazemos nosso teste esperar 5 segundos para garantir que o índice de busca do Solr tenha sido carregado. Nosso método de teste é definido usando a anotação `@Test` do *JUnit*:

```
@Test
public void searchScenario() throws IOException {
  ...
}
```

Primeiramente, verificamos que a página foi carregada corretamente, verificando o título da janela do navegador usando o método `getTitle()` do `driver`:

```
assertThat(driver.getTitle(),
           is("Broadleaf Demo - Heat Clinic"));
```

Sabendo que estamos na página correta, procuramos pelo elemento correspondente ao campo de busca com nome `q`, usando o método `findElement(By by)`. Preenchemos então o campo com o termo de busca *"hot sauce"* e submetemos o formulário usando o método `submit()`:

```
WebElement searchField = driver.findElement(By.name("q"));
searchField.sendKeys("hot sauce");
searchField.submit();
```

O método `submit()` também irá bloquear até que a nova página seja carregada. Feito isso, basta verificar que a busca retornou 18 produtos, validando o texto de descrição através do método `getText()`:

```
WebElement searchResults = driver.findElement(
            By.cssSelector("#left_column > header > h1"));
assertThat(searchResults.getText(),
           is("Search Results hot sauce (1 - 15 of 18)"));
```

Repare que dessa vez estamos usando um seletor CSS para encontrar o elemento desejado na página. Por fim, podemos também verificar que a página atual mostra apenas 15 resultados, agora usando o método `findElements(By by)` que, em vez de retornar um único elemento, retorna uma lista de elementos:

```
List<WebElement> results = driver.findElements(
            By.cssSelector("ul#products > li"));
assertThat(results.size(), is(15));
```

O *build* do Maven está configurado para executar nosso teste funcional como parte da fase `integration-test`, pois precisamos subir tanto o banco de dados quanto a aplicação antes de executar esse tipo de teste. Analisando a saída do comando `mvn install` executado anteriormente, você também irá encontrar o resultado dos testes funcionais no *build* do componente `site`:

```
...
[INFO] --- maven-surefire-plugin:2.10:test (integration-test)
           @ site ---
[INFO] Surefire report directory: ...

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running br.com.devopsnapratica.acceptance.SearchTest
...
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time ...

Results :
```

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
...
```

Com isso, demonstramos como integrar diferentes tipos de teste automatizados no processo de *build* do seu projeto. Agora temos os ingredientes básicos para discutir uma prática de desenvolvimento essencial para trabalhar em equipe de forma segura e eficiente: **integração contínua**.

6.5 O que é integração contínua?

Quando vários desenvolvedores estão trabalhando no mesmo sistema, aparecem novos desafios no processo de desenvolvimento de software. Trabalhar em paralelo exige mais comunicação e coordenação entre os membros da equipe pois quanto mais tempo eles ficarem sem integrar suas mudanças, maior o risco de criarem conflitos.

Integração contínua é uma das práticas originais de XP que encoraja desenvolvedores a integrar seu trabalho frequentemente para que possíveis problemas sejam detectados e corrigidos rapidamente. Assim que um desenvolvedor termina uma tarefa cujo código pode ser compartilhado com a equipe, ele precisa seguir um processo disciplinado para garantir que suas mudanças não vão introduzir problemas e que elas vão funcionar corretamente com o resto do código.

Ao terminar sua tarefa, você roda um *build* local executando todos os testes automatizados. Isso garante que as suas mudanças funcionam isoladamente e podem ser integradas com o resto do código. O próximo passo é atualizar sua cópia local do projeto, puxando as alterações mais recentes do repositório central. É preciso então executar os testes novamente para garantir que as suas mudanças funcionam junto com as alterações recentes que foram introduzidas por outros desenvolvedores. Só então é que você pode compartilhar seu *commit* com o repositório central.

Idealmente, cada desenvolvedor precisa fazer pelo menos um *commit* por dia para evitar o acúmulo de alterações locais que não estão visíveis para o resto da equipe. Equipes maduras têm o hábito de comitar ainda mais frequentemente, geralmente diversas vezes ao dia. No pior caso, se você está

trabalhando em uma mudança que vai demorar mais tempo e não tem como quebrá-la em pedaços menores, você deve atualizar sua cópia local do projeto o máximo possível para evitar a divergência com o que está acontecendo no repositório central.

Esse ritual diminui a chance de você introduzir uma mudança que quebra os testes do projeto. No entanto, exige também que todo desenvolvedor tenha a disciplina de fazer isso a cada *commit*. Por esse motivo, apesar de não ser estritamente necessário, é muito comum a equipe provisionar um servidor dedicado à integração contínua, responsável por garantir que o processo está sendo seguido corretamente.

O servidor de integração contínua é responsável por monitorar o repositório central de código e iniciar um *build* do projeto toda vez que detectar um novo *commit*. Ao final da execução de um *build*, ele pode passar ou falhar. Quando o *build* falha, é comum dizer que ele quebrou ou que "está vermelho". Quando o *build* termina com sucesso, é comum dizer que o *build* passou ou que "está verde".

O servidor de integração contínua funciona também como um radiador de informações da equipe, mantendo todos os desenvolvedores informados do estado atual do *build* do projeto. Ferramentas nessa área possuem diversas maneiras de disseminar essa informação: através de painéis de controle e interfaces web, enviando e-mails ou notificações para a equipe toda vez que o *build* falhar, ou expondo uma API que pode ser usada para mostrar o estado atual do projeto em outros dispositivos.

Quando um problema de integração é detectado e o *build* quebra, uma boa prática é que nenhum outro desenvolvedor compartilhe seus *commits* enquanto o *build* não for consertado. Geralmente os responsáveis pelo último *commit* começam a investigar o problema e decidem entre consertar o problema ou reverter o *commit* problemático. Essa política é boa para reforçar a importância de manter o *build* funcionando já que a equipe fica bloqueada para compartilhar novas mudanças enquanto o problema não é resolvido.

Existem diversas ferramentas comerciais e de código livre que podem ser usadas para configurar um servidor de integração contínua. Algumas das mais populares são o **Jenkins** (http://jenkins-ci.org/) , o **CruiseControl** (http://cruisecontrol.sourceforge.net/) , o **Go** (http://go.thoughtworks.com/)

, o **TeamCity** (http://www.jetbrains.com/teamcity/) , e o **Bamboo** (http://www.atlassian.com/software/bamboo) . Existem também serviços de integração contínua na nuvem no estilo SaaS (*Software as a Service* ou Software como Serviço) como, por exemplo, o **TravisCI** (http://travis-ci.org/) , o **SnapCI** (http://snap-ci.com/) ou o **Cloudbees** (http://cloudbees.com/) . Para uma lista mais completa das diferentes ferramentas de integração contínua, vale a pena visitar a página da Wikipedia http://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software.

Nosso objetivo não é entrar no mérito ou nas vantagens e desvantagens de cada ferramenta, mas sim mostrar um exemplo do processo de integração contínua funcionando na prática. Por isso escolhemos usar o Jenkins, que é uma ferramenta de código livre bem popular no mundo Java, possui um amplo ecossistema de plugins e uma comunidade bem ativa.

Antes de entrarmos nos detalhes de como instalar e configurar o servidor Jenkins, é recomendado que você tenha seu próprio repositório Git da loja virtual, onde você terá liberdade de fazer seus próprios *commits* para exercitarmos nosso processo de integração contínua. Para isso, você pode simplesmente criar seu próprio *fork* do repositório principal no GitHub. Um *fork* nada mais é que o seu próprio clone de qualquer repositório do GitHub. Esta funcionalidade existe para que membros da comunidade de software livre possam fazer modificações e contribuí-las de volta para o repositório principal.

Para criar o seu *fork* você vai precisar de um usuário cadastrado no GitHub. Feito isso, acesse a URL do repositório da loja virtual em https://github.com/dtsato/loja-virtual-devops e clique no botão *Fork*, conforme mostra a figura 6.2. Após alguns segundos, você terá uma cópia completa do repositório em uma nova URL que será parecida com a URL anterior, porém substituindo o usuário `dtsato` pelo seu usuário do GitHub.
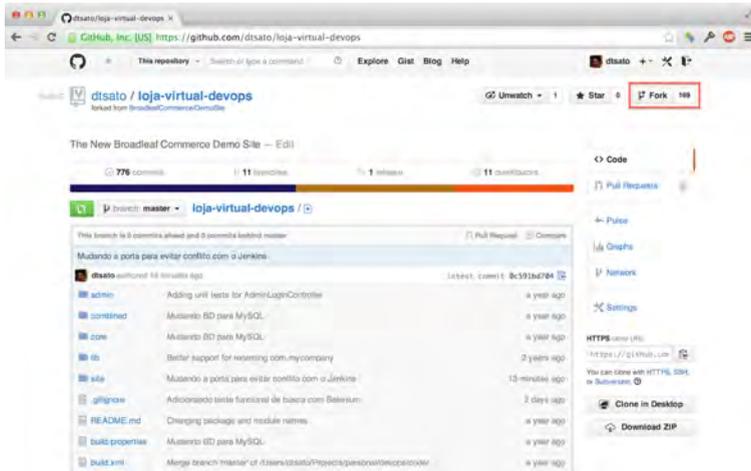
Figura 6.2: Fazendo fork do repositório no GitHub

Uma vez que o seu *fork* do repositório tiver sido criado, você pode executar o seguinte comando para obter uma cópia local na sua máquina, substituindo `<usuário>` pelo seu usuário do GitHub:

```
$ git clone https://github.com/<usuário>/loja-virtual-devops.git
Cloning into 'loja-virtual-devops'...
remote: Counting objects: 11358, done.
remote: Compressing objects: 100% (3543/3543), done.
remote: Total 11358 (delta 6053), reused 11358 (delta 6053)
Receiving objects: 100% (11358/11358), 55.47 MiB | 857.00 KiB/s,
                   done.
Resolving deltas: 100% (6053/6053), done.
$ cd loja-virtual-devops
```

6.6 Provisionando um servidor de integração contínua

Como já estamos familiarizados com o provisionamento usando Vagrant e Puppet, vamos adicionar uma nova máquina virtual chamada `ci` no arquivo `Vagrantfile`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...
  config.vm.define :db ...
  config.vm.define :web ...
  config.vm.define :monitor ...

  config.vm.define :ci do |build_config|
    build_config.vm.network :private_network,
                            :ip => "192.168.33.16"
    build_config.vm.provision "puppet" do |puppet|
      puppet.module_path = ["modules", "librarian/modules"]
      puppet.manifest_file = "ci.pp"
    end
  end
end
```

A nova máquina virtual será provisionada pelo Puppet usando um novo arquivo de manifesto chamado `ci.pp`. Você pode criar esse novo arquivo dentro do diretório `manifests` e seu conteúdo é bem simples:

```
include loja_virtual::ci
```

Seguindo o mesmo padrão que criamos no capítulo 5, vamos definir os recursos do servidor de integração contínua em uma nova classe chamada `loja_virtual::ci` dentro do módulo `loja_virtual`. Para isto, precisamos criar um novo arquivo no caminho `modules/loja_virtual/manifests/ci.pp` no qual essa nova classe é definida:

```
class loja_virtual::ci {
  include loja_virtual
  ...
}
```

É aqui que vem a parte interessante da instalação e configuração do Jenkins. Primeiramente precisamos instalar alguns pacotes básicos para as ferramentas de desenvolvimento e *build* como o Git, Maven e o compilador Java:

```
package { ['git', 'maven2', 'openjdk-6-jdk']:
  ensure => "installed",
}
```

Para instalar e configurar o Jenkins vamos usar um módulo escrito pela comunidade, cuja documentação pode ser encontrada em https://github.com/jenkinsci/puppet-jenkins. Este módulo define uma classe chamada `jenkins` que instala um servidor Jenkins com configurações padrão. Para usá-la, basta adicionar o recurso:

```
class { 'jenkins':
  config_hash => {
    'JAVA_ARGS' => { 'value' => '-Xmx256m' }
  },
}
```

Estamos passando um *hash* que irá sobrescrever a configuração padrão de uma das opções passadas na inicialização do Jenkins. Definindo a configuração `JAVA_ARGS` com o valor `"-Xmx256m"` estamos dizendo que o processo Java do Jenkins pode alocar e usar até 256Mb de memória.

Um dos principais atrativos do Jenkins é seu ecossistema de plugins. Para uma lista completa de todos os plugins suportados pelo Jenkins você pode acessar sua documentação em https://wiki.jenkins-ci.org/display/JENKINS/Plugins. O módulo Puppet possui um tipo definido que permite instalar diversos plugins:

```
$plugins = [
  'ssh-credentials',
  'credentials',
  'scm-api',
  'git-client',
  'git',
  'javadoc',
  'mailer',
  'maven-plugin',
  'greenballs',
  'ws-cleanup'
]
```

```
jenkins::plugin { $plugins: }
```

No nosso caso, os principais plugins que estamos interessados são:

- **Plugin `git`** (http://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin) : permite que o Jenkins monitore e inicie *builds* a partir de um repositório Git ao invés do padrão SVN. Os plugins `ssh-credentials`, `credentials`, `scm-api` e `git-client` são dependências desse plugin, por isso precisamos incluí-los.
- **Plugin `maven-plugin`** (http://wiki.jenkins-ci.org/display/JENKINS/Maven+Project+Plugin) : permite que o Jenkins execute um *build* do Maven. Os plugins `javadoc` e `mailer` são dependências desse plugin, por isso precisamos incluí-los.
- **Plugin `greenballs`** (http://wiki.jenkins-ci.org/display/JENKINS/Green+Balls) : alteração cosmética na interface web do Jenkins para mostrar *builds* bem sucedidos na cor verde em vez do padrão azul.
- **Plugin `ws-cleanup`** (https://wiki.jenkins-ci.org/display/JENKINS/Workspace+Cleanup+Plugin) : permite que o Jenkins limpe os arquivos entre a execução de um *build* e outro.

Com isso, a classe `loja_virtual::ci` completa fica assim:

```
class loja_virtual::ci {
  include loja_virtual

  package { ['git', 'maven2', 'openjdk-6-jdk']:
    ensure => "installed",
  }

  class { 'jenkins':
    config_hash => {
      'JAVA_ARGS' => { 'value' => '-Xmx256m' }
    },
  }
```

```
  $plugins = [
    'ssh-credentials',
    'credentials',
    'scm-api',
    'git-client',
    'git',
    'maven-plugin',
    'javadoc',
    'mailer',
    'greenballs',
    'ws-cleanup'
  ]

  jenkins::plugin { $plugins: }
}
```

Podemos agora tentar subir o servidor `ci` usando o Vagrant, porém encontramos um problema:

```
$ vagrant up ci
Bringing machine 'ci' up with 'virtualbox' provider...
==> ci: Importing base box 'hashicorp/precise32'...
==> ci: Matching MAC address for NAT networking...
...
Puppet::Parser::AST::Resource failed with error ArgumentError:
Could not find declared class jenkins at
/tmp/vagrant-puppet-2/modules-0/loja_virtual/manifests/ci.pp:12
on node precise32
```

A classe `jenkins` não foi encontrada pois esquecemos de adicioná-la no arquivo `Puppetfile` que o Librarian Puppet usa para instalar os módulos do Puppet Forge. Para isto, alteramos o arquivo `librarian/Puppetfile` adicionando o módulo `rtyler/jenkins` na versão `1.0.0`:

```
forge "http://forge.puppetlabs.com"

mod "puppetlabs/apt", "1.4.0"
mod "rtyler/jenkins", "1.0.0"
```

Agora podemos reprovisionar o servidor `ci` e o Jenkins será instalado com sucesso:

```
$ vagrant provision ci
==> ci: Installing Puppet modules with Librarian-Puppet...
==> ci: Running provisioner: puppet...
Running Puppet with ci.pp...
...
notice: Finished catalog run in 84.19 seconds
```

Para verificar que o Jenkins foi instalado e configurado corretamente, você pode abrir uma nova janela no seu navegador web e acessar a URL http://192.168.33.16:8080/. Se tudo estiver certo você verá a tela de boas-vindas do Jenkins mostrada na figura 6.3.

Figura 6.3: Tela de boas-vindas do Jenkins

6.7 Configurando o build da loja virtual

Agora que o Jenkins está instalado e rodando, precisamos configurá-lo para rodar o *build* da loja virtual toda vez que houver um *commit* no repositório Git. No entanto, antes de fazer isso, precisamos fazer uma configuração manual para que o Jenkins reconheça onde o Maven está instalado no sistema.

Para isto, você deve clicar no link *"Manage Jenkins"* na tela principal, e na tela seguinte clique no primeiro link *"Configure System"*, que nos levará para a tela de administração do Jenkins mostrada na figura 6.4.
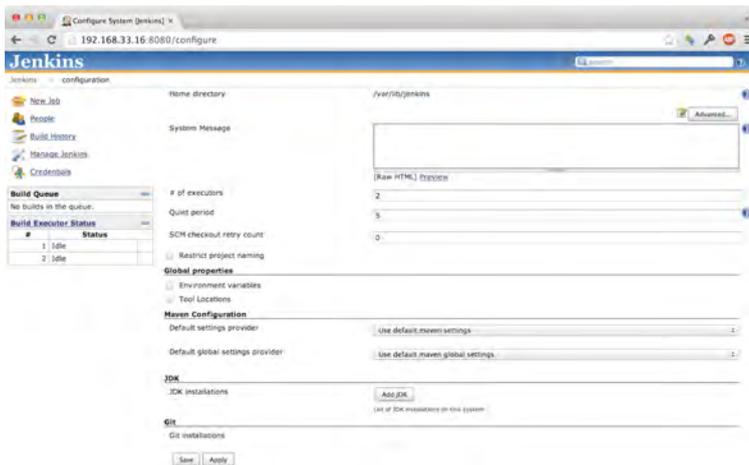
Figura 6.4: Tela de administração do Jenkins

Nesta tela, um pouco mais para baixo, você verá uma seção chamada *"Maven"* com uma opção *"Maven installations"*. Clique no botão *"Add Maven"* e ele irá expandir novas opções. Iremos então desmarcar a opção *"Install automatically"* pois o Maven já está instalado no sistema. Precisamos apenas dizer para o Jenkins o caminho onde o Maven pode ser encontrado, preenchendo o campo *"Name"* com o valor *"Default"* e o campo *"MAVEN_HOME"* com o valor *"/usr/share/maven2"*. A figura 6.5 mostra como deve ficar a sua configuração após seguir estas instruções, destacando em vermelho as áreas importantes.
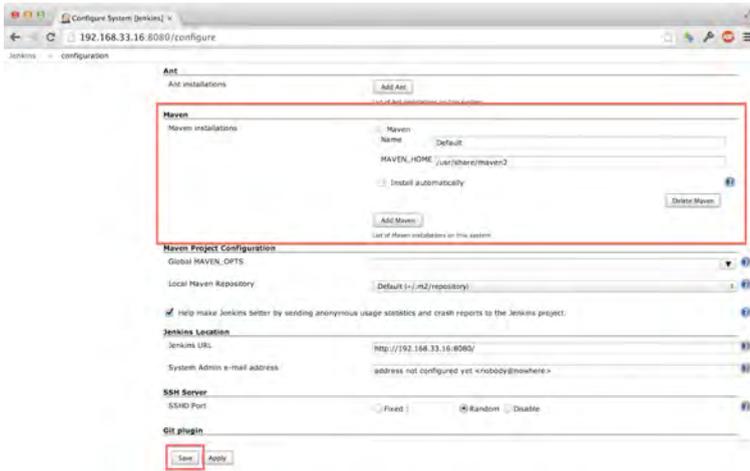
Figura 6.5: Configurando o Maven no Jenkins

É preciso então clicar no botão *"Save"* na parte inferior da página para que as alterações tenham efeito. Isso irá nos levar de volta à tela de boas-vindas do Jenkins. Podemos então criar o *build* da loja virtual clicando no link *"create new jobs"*. Na terminologia do Jenkins um ***job*** – ou **trabalho** – representa a tarefa que você quer executar repetidamente. Nele, você configura os parâmetros de execução e as tarefas do seu *build*. O *job* armazena o histórico de execuções assim como o estado atual do seu *build*.

Uma nova tela será exibida para você definir o nome do *job* e seu tipo. Iremos definir o nome como *"loja-virtual-devops"* e escolheremos a opção *"Build a maven2/3 project"* conforme mostra a figura 6.6.
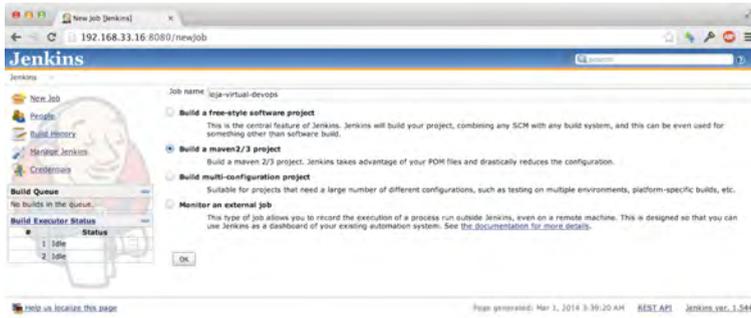
Figura 6.6: Criando um novo job no Jenkins

Ao clicar no botão *"OK"*, chegamos na tela principal de configuração do novo *job*. Vamos descrever passo a passo o que precisamos fazer em cada seção:

- **Seção *"Source Code Management"***: nesta seção definimos qual sistema de controle de versões o Jenkins deve usar para obter o código do projeto. Ao escolher a opção *"Git"*, novas opções serão expandidas. A única opção importante é *"Repository URL"* onde você deve preencher com a URL do seu repositório no GitHub criado no final da seção 6.5, conforme mostra a figura 6.7.
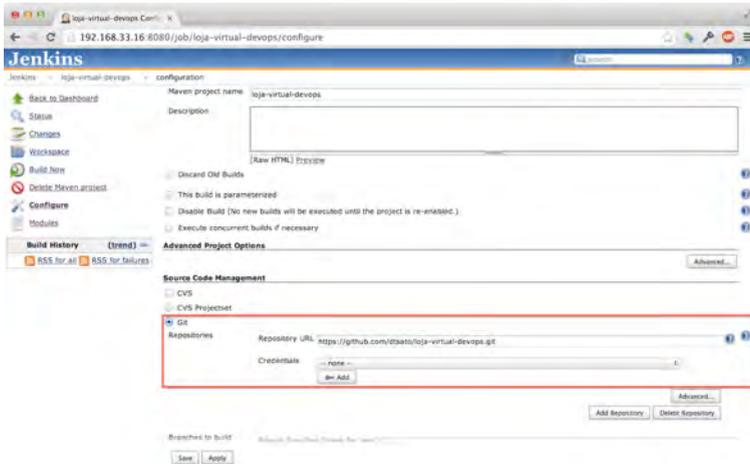
Figura 6.7: Configurando o sistema de controle de versões

- **Seção *"Build Triggers"***: nesta seção definimos quais eventos irão dar início a uma nova execução do *build* do projeto. Iremos desmarcar a opção *"Build whenever a SNAPSHOT dependency is built"* e escolher a opção *"Poll SCM"* que significa monitorar o sistema de controle de versões. Uma nova caixa de texto *"Schedule"* será expandida, na qual você precisa definir a frequência com que o Jenkins irá verificar se existe algum novo *commit* no repositório. Iremos preenchê-la com o valor *"* * * * *"*, que significa "monitore mudanças a cada minuto", conforme mostra a figura 6.8
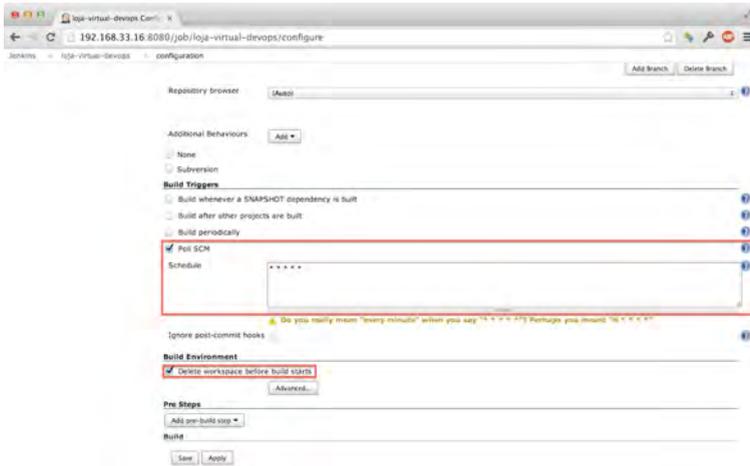
Figura 6.8: Configurando gatilho e limpeza da área de trabalho

- **Seção *"Build Environment"***: nesta seção vamos escolher a opção *"Delete workspace before build starts"* para que o Jenkins sempre limpe a área de trabalho antes de começar um novo *build*, conforme mostra a figura 6.8. Isso garante que o *build* sempre execute de forma determinística, pois nenhum arquivo vai ficar sobrando entre uma execução e outra.

- **Seção *"Build"***: nesta seção definimos qual comando deve ser executado para rodar o *build*. Como estamos usando o plugin do Maven, só precisamos preencher o campo *"Goals and options"* com o valor *"install"*, conforme mostra a figura 6.9.
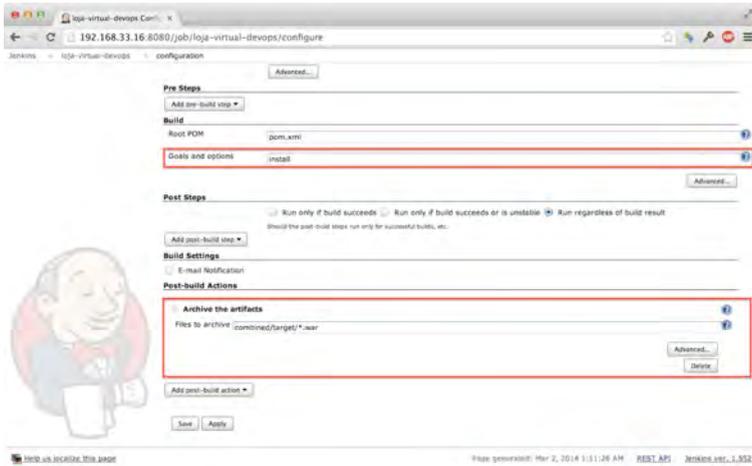
Figura 6.9: Tarefa do Maven e arquivando artefatos no final do build

- **Seção *"Post-build Actions"***: nesta seção definimos o que deve acontecer após a execução do *build*. Ao clicar no botão *"Add post-build action"*, iremos escolher a opção *"Archive the artifacts"*, que diz ao Maven para arquivar os artefatos gerados pelo processo de *build*. Novas opções serão expandidas e basta preencher o campo *"Files to archive"* com o valor *"combined/target/*.war"*, conforme mostra a figura 6.9. Isso fará com que o Jenkins arquive o artefato `.war` gerado no final do *build* para que possamos obtê-lo mais para frente, na hora de fazer um *deploy*.

Ao final de todas essas configurações, você pode clicar no botão *"Save"* no final da página e o *job* será criado e irá começar a monitorar o repositório Git imediatamente. Em menos de um minuto você verá que o Jenkins vai agendar a execução do primeiro *build* do seu projeto e, quando o *build* número 1 começar a rodar, você verá uma barra de progresso no canto inferior direito, conforme mostra a figura 6.10.
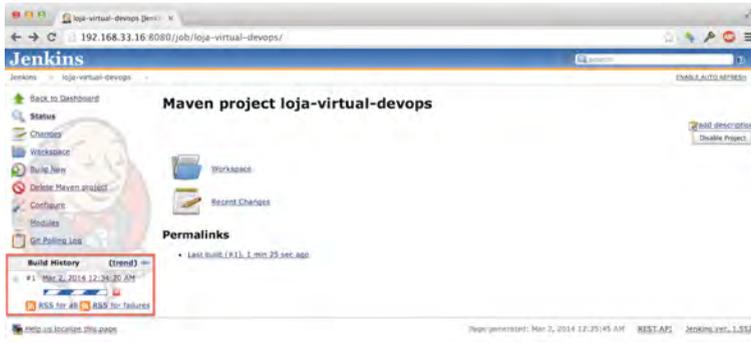
Figura 6.10: Primeiro build agendado para execução

O primeiro *build* geralmente demora um pouco mais pois o Maven precisa resolver e baixar todas as dependências do projeto. Para acompanhar o progresso e ver a saída como se estivesse rodando o *build* em um terminal, você pode clicar na barra de progresso. Esta página se atualiza automaticamente e, ao final da execução do *build*, você verá a mensagem *"Finished: SUCCESS"* que nos informa que o *build* passou! A figura 6.11 mostra um exemplo de um *build* que terminou com sucesso.
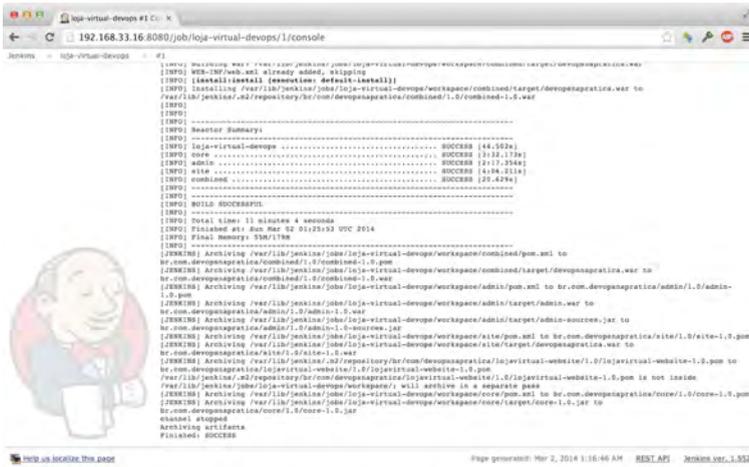
Figura 6.11: Build finalizado com sucesso

Para um resumo da execução, acesse a URL do *job* em http://192.168.33.16:8080/job/loja-virtual-devops/ e você verá uma página parecida com a figura 6.12. No canto inferior esquerdo, você verá o histórico dos *builds* do projeto e a bolinha verde representa sucesso – quando o *build* falha, ela fica vermelha – e no centro da página você verá um link para o artefato `.war` gerado no último *build* bem sucedido.

Figura 6.12: Resumo da execução e artefato arquivado

Agora temos um servidor de integração contínua que roda um *build* toda vez que detectar um *commit* no repositório Git. Para testar a instalação um pouco mais, vamos criar um novo *commit* com uma pequena alteração e validar que o Jenkins roda um novo *build* do projeto. Para isto, vamos usar o repositório que você clonou do GitHub na seção anterior. Dentro do diretório do projeto clonado, execute o comando:

```
$ echo -e "\n" >> README.md
```

O comando `echo` adiciona uma quebra de linha no final do arquivo `README.md`. Caso você esteja rodando este exemplo no Windows e o comando não funcione, você pode simplesmente alterar o arquivo no seu editor de texto preferido e salvá-lo.

A seguir, o comando `git commit` cria um novo *commit* com uma mensagem explicando o que mudou e o comando `git push` envia o *commit* local para o repositório central no GitHub. Neste último comando, você pode precisar fornecer seu usuário e senha do GitHub, dependendo de como sua conta está configurada.

```
$ git commit -am"Adicionando quebra de linha ao final do README"
[master 4d191bd] Adicionando quebra de linha ao final do README
```

```
 1 file changed, 2 insertions(+), 1 deletion(-)
$ git push origin master
Counting objects: 15, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 310 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:dtsato/loja-virtual-devops.git
   9ae96a8..4d191bd  master -> master
```

Assim que o Jenkins detectar o novo *commit*, um novo *build* será agendado, conforme mostra a figura 6.13.
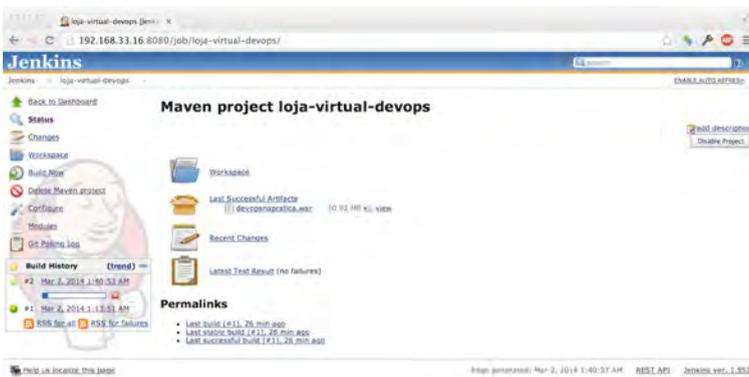
Figura 6.13: Novo build agendado após um novo commit

Dessa vez, a execução do *build* deve ser mais rápida. No nosso exemplo, enquanto o primeiro *build* demorou aproximadamente 12 minutos, o segundo durou apenas 4 minutos, pois todas as dependências já estavam no *cache* local do Maven. Ao final da execução, você pode novamente ir para a página do *job* em http://192.168.33.16:8080/job/loja-virtual-devops/ e ver não só o mais novo artefato disponível como também um gráfico mostrando o histórico de execução dos testes do projeto, conforme mostra a figura 6.14.
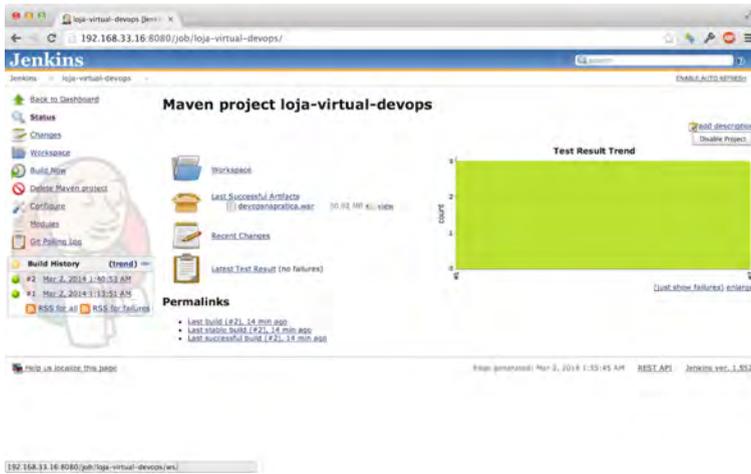
Figura 6.14: Mais um build bem-sucedido

6.8 Infraestrutura como código para o servidor de integração contínua

Com o Jenkins instalado e funcionando corretamente, poderíamos acabar o capítulo por aqui. No entanto, vocês devem ter percebido que fizemos diversas configurações manuais através da interface web do Jenkins e isso pode ser perigoso se houver problemas com nosso servidor de integração contínua.

Quando uma equipe passa a praticar integração contínua, esse servidor passa a ser parte essencial da infraestrutura do projeto. Da mesma forma que nenhum desenvolvedor pode fazer um *commit* quando o *build* está quebrado, a equipe inteira fica bloqueada quando o servidor de integração contínua está fora do ar.

Por esse motivo precisamos ser capazes de trazer ao ar um novo servidor de integração contínua completamente configurado o mais rápido possível. Iremos um pouco mais além para codificar todas as etapas manuais em código Puppet para que possamos provisionar um novo servidor de integração contínua completo em caso de emergência.

O Jenkins armazena a maioria de suas configurações internas em ar-

quivos XML dentro do diretório /var/lib/jenkins. A primeira configuração manual que fizemos na última seção foi dizer ao Jenkins onde o Maven está instalado no sistema. Essa configuração é salva no arquivo /var/lib/jenkins/hudson.tasks.Maven.xml. Vamos usar o mesmo truque do capítulo 4 para copiar o arquivo de dentro da máquina virtual ci para nossa máquina local:

```
$ vagrant ssh ci -- 'sudo cp /var/lib/jenkins/
                                        hudson.tasks.Maven.xml \
> /vagrant/modules/loja_virtual/files'
```

Esse comando irá salvar o arquivo dentro do módulo Puppet da loja_virtual em modules/loja_virtual/files. Precisamos adicionar um novo recurso na classe loja_virtual::ci, alterando o arquivo modules/loja_virtual/manifests/ci.pp:

```
class loja_virtual::ci inherits loja_virtual {
  package { ... }
  class { 'jenkins': ... }
  $plugins = ...
  jenkins::plugin { ... }

  file { '/var/lib/jenkins/hudson.tasks.Maven.xml':
    mode    => 0644,
    owner   => 'jenkins',
    group   => 'jenkins',
    source  => 'puppet:///modules/loja_virtual/
                                        hudson.tasks.Maven.xml',
    require => Class['jenkins::package'],
    notify  => Service['jenkins'],
  }
}
```

A próxima configuração manual que fizemos foi criar o *job*. Sua configuração fica armazenada no arquivo /var/lib/jenkins/jobs/loja-virtual-devops/config.xml. Vamos usar o mesmo truque para copiar esse arquivo de dentro da máquina virtual para um novo diretório de *templates* no módulo Puppet da loja_virtual:

```
$ mkdir modules/loja_virtual/templates
$ vagrant ssh ci -- 'sudo cp \
> /var/lib/jenkins/jobs/loja-virtual-devops/config.xml \
> /vagrant/modules/loja_virtual/templates'
```

Para instruir o Puppet como criar o *job* após a instalação do Jenkins, vamos precisar definir algumas variáveis com as configurações necessárias dentro do arquivo `ci.pp`. Não se esqueça de substituir o usuário `dtsato` pelo seu usuário do GitHub na URL da variável `$git_repository`:

```
class loja_virtual::ci inherits loja_virtual {
  package { ... }
  class { 'jenkins': ... }
  $plugins = ...
  jenkins::plugin { ... }
  file { '/var/lib/jenkins/hudson.tasks.Maven.xml': ... }

  $job_structure = [
    '/var/lib/jenkins/jobs/',
    '/var/lib/jenkins/jobs/loja-virtual-devops'
  ]
  $git_repository = 'https://github.com/dtsato/
                                        loja-virtual-devops.git'
  $git_poll_interval = '* * * * *'
  $maven_goal = 'install'
  $archive_artifacts = 'combined/target/*.war'
}
```

Precisamos então substituir os valores que estão *hardcoded* dentro do arquivo XML por expressões ERB que usam as variáveis que acabamos de criar. Os elementos do XML que precisam ser alterados no arquivo `modules/loja_virtual/templates/config.xml` estão em destaque a seguir, enquanto as partes não alteradas mostram apenas reticências (...):

```
<?xml version='1.0' encoding='UTF-8'?>
<maven2-moduleset plugin="maven-plugin@2.1">
  ...
  <scm class="hudson.plugins.git.GitSCM" plugin="git@2.0.3">
    ...
```

```
    <userRemoteConfigs>
      <hudson.plugins.git.UserRemoteConfig>
        <url><%= git_repository %></url>
      </hudson.plugins.git.UserRemoteConfig>
    </userRemoteConfigs>
    ...
  </scm>
  ...
  <triggers>
    <hudson.triggers.SCMTrigger>
      <spec><%= git_poll_interval %></spec>
      ...
    </hudson.triggers.SCMTrigger>
  </triggers>
  ...
  <goals><%= maven_goal %></goals>
  ...
  <publishers>
    <hudson.tasks.ArtifactArchiver>
      <artifacts><%= archive_artifacts %></artifacts>
      ...
    </hudson.tasks.ArtifactArchiver>
  </publishers>
  ...
</maven2-moduleset>
```

Com o *template* criado, precisamos alterar a classe `loja_virtual::ci` para criar um novo diretório para o nosso *job* e usar o *template* para criar o arquivo `config.xml`. Faremos isto adicionando dois novos recursos em `modules/loja_virtual/manifests/ci.pp`:

```
class loja_virtual::ci inherits loja_virtual {
  package { ... }
  class { 'jenkins': ... }
  $plugins = ...
  jenkins::plugin { ... }
  file { '/var/lib/jenkins/hudson.tasks.Maven.xml': ... }
  $job_structure = ...
  $git_repository = ...
```

```
  $git_poll_interval = ...
  $maven_goal = ...
  $archive_artifacts = ...

  file { $job_structure:
    ensure  => 'directory',
    owner   => 'jenkins',
    group   => 'jenkins',
    require => Class['jenkins::package'],
  }

  file { "${job_structure[1]}/config.xml":
    mode    => 0644,
    owner   => 'jenkins',
    group   => 'jenkins',
    content => template('loja_virtual/config.xml'),
    require => File[$job_structure],
    notify  => Service['jenkins'],
  }
}
```

O recurso `File[$job_structure]` garante que a estrutura de diretórios do *job* seja criada enquanto o recurso `File["${job_structure[1]}/config.xml"]` usa o *template* ERB para criar o arquivo de configuração do *job* e notificar o serviço do Jenkins para que ele seja reiniciado.

Com essas alterações, podemos testar se o Puppet realmente consegue recriar o servidor de integração contínua do zero. Faremos isto usando o comando `vagrant destroy ci` para derrubar o servidor `ci`, seguido de um comando `vagrant up ci` que irá reprovisioná-lo:

```
$ vagrant destroy ci
    ci: Are you sure you want to destroy the 'ci' VM? [y/N] y
==> ci: Forcing shutdown of VM...
==> ci: Destroying VM and associated drives...
==> ci: Running cleanup tasks for 'puppet' provisioner...
$ vagrant up ci
Bringing machine 'ci' up with 'virtualbox' provider...
```

```
==> ci: Importing base box 'hashicorp/precise32'...
==> ci: Matching MAC address for NAT networking...
...
notice: Finished catalog run in 76.70 seconds
```

Se tudo der certo, ao final da execução do Puppet, você pode acessar a interface web do Jenkins em http://192.168.33.16:8080/ e, em vez de ver a página de boas-vindas, você verá que o *job* da loja virtual já existe e provavelmente estará rodando, conforme mostra a figura 6.15.
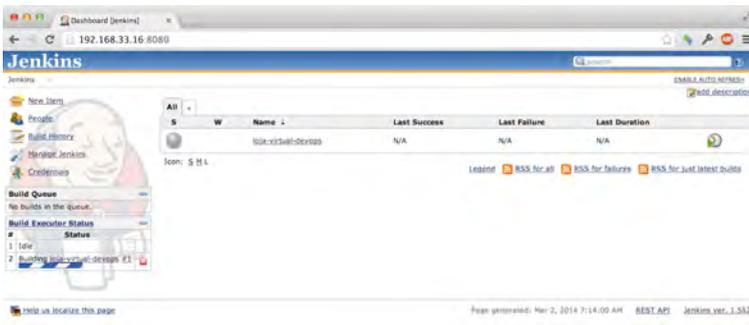
Figura 6.15: Jenkins recriado do zero e rodando um novo build

Neste capítulo aprendemos um pouco mais sobre algumas práticas de engenharia ágil que permitem desenvolver software de qualidade: sistemas de controle de versão, processos de *build* automatizados, alguns dos principais tipos de testes automatizados e, por fim, como provisionar um servidor de integração contínua para detectar problemas o mais rápido possível e garantir que a qualidade do software está sendo avaliada a cada *commit*.

Além disso, encontramos uma forma melhor de gerar o artefato `.war` ao final de cada *build* fora dos servidores de produção. No entanto, o artefato está publicado apenas dentro do Jenkins. Precisamos de uma maneira de unir nosso *build* com o processo de *deploy* para darmos o próximo passo a

caminho da entrega contínua.

Capítulo 7

Pipeline de entrega

Agora que adotamos a prática de integração contínua, temos uma forma confiável de gerar e validar novas versões da loja virtual a cada *commit*. Ao final de um *build* bem sucedido temos um artefato `.war` que vira candidato a ir para produção. Esse é um passo importante na jornada para implementar entrega contínua e aumentar a frequência de *deploys*. Temos os componentes necessários para criar um processo de *deploy* que acontece ao clique de um botão, só precisamos ligar os pontos.

Atualmente, o módulo Puppet da `loja_virtual` possui um arquivo `.war` com uma versão fixa criada quando fizemos o primeiro *build* manual no capítulo 2. Para usar um arquivo `.war` mais recente, poderíamos baixar uma cópia local acessando a página do *job* no Jenkins, colocá-lo dentro do módulo Puppet e reprovisionar o servidor `web`. Em vez de fazer isso de forma manual vamos aprender como compartilhar artefatos usando um repositório de pacotes que pode ser diretamente acessado pelo nosso código

Puppet durante um *deploy*.

Além disso, vamos discutir como integrar o código de infraestrutura no nosso processo automatizado de entrega e como modelar as diferentes etapas necessárias para levar código de um *commit* até produção com qualidade e confiança.

7.1 Afinidade com a infraestrutura: usando pacotes nativos

Copiar arquivos para lá e para cá não é a forma mais eficiente de fazer *deploy*. É por isso que no mundo Java é comum usar arquivos `.war` ou `.jar` para agrupar diversos arquivos em um único pacote. Eles nada mais são que arquivos `ZIP` – um formato de compactação bem conhecido – com alguns metadados extras que indicam o conteúdo do pacote. Ecossistemas de outras linguagens também possuem seus próprios formatos de empacotamento e distribuição: `.gem` em Ruby, `.dll` em .NET etc.

Administradores de sistema estão acostumados a usar o sistema de pacotes nativo do sistema operacional para empacotar, distribuir e instalar software. Nós inclusive já os estamos usando extensivamente no nosso código Puppet, toda vez que criamos um recurso do tipo `Package` ou o comando `apt-get` para instalar o MySQL, o Tomcat, o Nagios etc.

Esse é um exemplo em que desenvolvedores e administradores de sistema possuem opiniões e ferramentas diferentes. DevOps pode melhorar essa colaboração simplesmente alinhando as ferramentas usadas durante o processo. Existem diversos motivos pelos quais administradores de sistema preferem usar pacotes nativos para instalar software:

- **Versionamento embutido e gerenciamento de dependências**: esse ponto é discutível pois alguns formatos – como Rubygems – possuem esse tipo de suporte. Arquivos `.jar` e `.war` também dão suporte a declarar a versão do pacote dentro do arquivo `META-INF/MANIFEST.MF`, porém isso não é obrigatório e não possui nenhuma semântica especial que as ferramentas possam aproveitar. Pacotes nativos, por outro lado, tratam versões e dependências como parte integrante do pacote e sabem resolvê-las na hora da instalação.

- **Sistema de distribuição**: repositórios são a forma natural de armazenar e compartilhar pacotes nativos e suas ferramentas de gerenciamento e instalação sabem realizar buscas e baixar os pacotes necessários na hora da instalação.

- **Instalação é transacional e idempotente**: gerenciadores de pacote dão suporte à instalação, desinstalação e atualização (ou *upgrade*) de pacotes e essas operações são transacionais e idempotentes. Você não corre o risco de instalar apenas metade do pacote e deixar arquivos soltos no sistema.

- **Suporte a arquivos de configuração**: pacotes nativos sabem identificar arquivos de configuração que podem ser alterados após a instalação. O gerenciador de pacotes irá manter o arquivo alterado ou irá guardar uma cópia do arquivo para que você não perca suas alterações quando fizer um *upgrade* ou quando remover o pacote, respectivamente.

- **Verificação de integridade**: quando pacotes são criados, um *checksum* é calculado com base no conteúdo do arquivo. Após o pacote ser baixado para instalação, o gerenciador de pacotes irá recalcular esse *checksum* e compará-lo com o que foi publicado no repositório para garantir que o pacote não tenha sido alterado ou corrompido no processo de *download*.

- **Verificação de assinatura**: da mesma forma, pacotes são criptograficamente assinados quando publicados no repositório. No processo de instalação, o gerenciador de pacotes pode verificar essa assinatura para garantir que o pacote está realmente vindo do repositório desejado.

- **Auditoria e rastreabilidade**: gerenciadores de pacotes permitem descobrir qual pacote instalou qual arquivo no sistema. Além disso você pode saber de onde um certo pacote veio e quem foi o responsável por criá-lo.

- **Afinidade com ferramentas de infraestrutura**: por fim, a maioria das ferramentas de automação de infraestrutura – como o Puppet, Chef,

Ansible, Salt etc.– sabem lidar com pacotes nativos, gerenciadores de pacote e seus repositórios.

Por esse motivo vamos aprender como criar um pacote nativo para a nossa aplicação. Além disso vamos criar e configurar um repositório de pacotes para publicar e distribuir esses novos pacotes gerados ao final de cada *build* bem sucedido.

Provisionando o repositório de pacotes

Nossos servidores são máquinas virtuais rodando Linux como sistema operacional. Em particular, estamos usando o Ubuntu, uma distribuição Linux baseada no Debian. Nessa plataforma, pacotes nativos são conhecidos como pacotes `.deb` e o `APT` é a ferramenta padrão para gerenciamento de pacotes.

Um repositório de pacotes nada mais é que uma estrutura de diretórios e arquivos bem definida. O repositório pode ser exposto de diversas maneiras, como por exemplo: HTTP, FTP, um sistema de arquivos local ou até mesmo um CD-ROM que costumava ser a forma mais comum de distribuição do Linux.

Vamos usar o **Reprepro** (http://mirrorer.alioth.debian.org/) para criar e gerenciar nosso repositório de pacotes. Utilizaremos o servidor `ci` para distribuir os pacotes pois podemos usar a mesma ferramenta para gerenciar o repositório e para publicar novos pacotes ao final de cada *build*. Para isso criaremos uma nova classe no nosso módulo `loja_virtual` chamada `loja_virtual::repo` dentro de um novo arquivo `modules/loja_virtual/manifests/repo.pp` com o seguinte conteúdo inicial:

```
class loja_virtual::repo($basedir, $name) {
  package { 'reprepro':
    ensure => 'installed',
  }
}
```

Isso irá instalar o pacote do Reprepro. Essa classe recebe dois parâmetros: `$basedir` será o caminho completo para o diretório

onde o repositório local será criado e `$name` será o nome do repositório. Precisamos também incluir essa classe no servidor `ci` e faremos isso alterando a classe `loja_virtual::ci` no arquivo `modules/loja_virtual/manifests/ci.pp`:

```
class loja_virtual::ci {
  ...
  $archive_artifacts = 'combined/target/*.war'
  $repo_dir = '/var/lib/apt/repo'
  $repo_name = 'devopspkgs'

  file { $job_structure: ... }
  file { "${job_structure[1]}/config.xml": ... }

  class { 'loja_virtual::repo':
    basedir => $repo_dir,
    name    => $repo_name,
  }
}
```

Adicionamos duas novas variáveis para representar o diretório raiz e o nome do repositório e criamos um novo recurso `Class['loja_virtual::repo']` passando essas variáveis como parâmetros da classe.

No caso do Ubuntu, cada versão do sistema operacional possui um codinome diferente, também conhecido como **distribuição**. No nosso caso, estamos usando o Ubuntu 12.04, também conhecido como `precise`. Repositórios Debian e Ubuntu também são divididos em **componentes**, que servem para comunicar os diferentes níveis de suporte: `main` contém software que é oficialmente suportado e livre; `restricted` possui software que é suportado porém com uma licença mais restritiva; `universe` contém pacotes mantidos pela comunidade em geral; `multiverse` contém software que não é livre. No nosso caso, como vamos distribuir um único pacote, todas essas classificações não importam tanto, então vamos distribuir nosso pacote como um componente `main`.

Para que o Reprepro crie a estrutura inicial de diretórios e arquivos do repositório, precisamos criar um arquivo de configuração chamado

distributions dentro de um diretório conf no diretório raiz do repositório. Para isso, vamos criar um novo arquivo de *template* ERB em modules/loja_virtual/templates/distributions.erb com o seguinte conteúdo:

```
Codename: <%= name %>
Architectures: i386
Components: main
SignWith: default
```

Nesse arquivo, Codename é a forma como a ferramenta da linha de comando do Reprepro se refere ao repositório por nome. Usamos a sintaxe de substituição do ERB `<%= name %>` que se refere ao parâmetro $name da definição da classe loja_virtual::repo.

A configuração Architectures lista quais arquiteturas de processador os pacotes desse repositório suporta. Quando um pacote possui arquivos compilados para linguagem de máquina, o processo de compilação usa um conjunto de instruções diferentes dependendo de onde o programa será executado, por isso é importante definir qual arquitetura foi usada na compilação para que o pacote execute corretamente após a instalação. No nosso caso, as máquinas virtuais estão rodando Ubuntu em um processador do tipo i386.

A configuração SignWith diz ao Reprepro como ele deve assinar os pacotes publicados nesse repositório. Usaremos o valor default que é a configuração padrão e discutiremos mais sobre assinatura de pacotes um pouco mais adiante.

O próximo passo é criar o diretório raiz do repositório e definir um novo recurso do tipo File para criar o arquivo de configuração distributions usando o *template* recém-criado. Faremos isso no arquivo modules/loja_virtual/manifests/repo.pp:

```
class loja_virtual::repo($basedir, $name) {
  package { 'reprepro':
    ensure => 'installed',
  }

  $repo_structure = [
    "$basedir",
```

```
    "$basedir/conf",
  ]

  file { $repo_structure:
    ensure  => 'directory',
    owner   => 'jenkins',
    group   => 'jenkins',
    require => Class['jenkins'],
  }

  file { "$basedir/conf/distributions":
    owner  => 'jenkins',
    group  => 'jenkins',
    content => template('loja_virtual/distributions.erb'),
    require => File["$basedir/conf"],
  }
}
```

A variável `$repo_structure` contém a lista de diretórios que precisa ser criada. Inicialmente você só precisa do diretório raiz e do diretório `conf`. Com isso podemos definir o recurso `File["$basedir/conf/distributions"]` que usará o *template* ERB para criar o arquivo de configuração do Reprepro.

Com o repositório criado, precisamos expor seus arquivos para que ele possa ser acessado de outros servidores. Usaremos o servidor web **Apache** (http://httpd.apache.org/) para que nosso repositório seja acessado por HTTP. Instalar e configurar o Apache é bem simples se usarmos o módulo `apache` do Puppet Forge e mantido pela própria PuppetLabs. Antes de mais nada, vamos adicionar a dependência ao novo módulo na versão `1.0.1` no arquivo `librarian/Puppetfile`:

```
forge "http://forge.puppetlabs.com"

mod "puppetlabs/apt", "1.4.0"
mod "rtyler/jenkins", "1.0.0"
mod "puppetlabs/apache", "1.0.1"
```

Com isso, basta modificar o arquivo `modules/loja_virtual/manifests/repo.pp` para adicionar a

classe `apache` e usar o tipo definido `apache::vhost` para configurar um *host* virtual:

```
class loja_virtual::repo($basedir, $name) {
  package { 'reprepro': ... }
  $repo_structure = [...]
  file { $repo_structure: ... }
  file { "$basedir/conf/distributions": ... }

  class { 'apache': }

  apache::vhost { $name:
    port       => 80,
    docroot    => $basedir,
    servername => $ipaddress_eth1,
  }
}
```

No Apache, um ***host* virtual** é a forma de rodar mais de um website no mesmo servidor. Os parâmetros `servername` e `port` definem o endereço e a porta onde um cliente pode acessar o *host* virtual. O parâmetro `docroot` define qual diretório será exposto nesse endereço. No nosso caso, vamos expor o diretório `$basedir` na porta 80 e usaremos o endereço IP do servidor para acessar o repositório.

A variável `$ipaddress_eth1` não está definida nessa classe e não é passada como parâmetro. Essa variável é fornecida pelo Facter, uma ferramenta utilitária do Puppet que expõe informações sobre o servidor para uso em qualquer arquivo de manifesto. Para ver todas as variáveis disponíveis você pode rodar o comando `facter` dentro de qualquer uma das máquinas virtuais:

```
$ vagrant ssh ci
vagrant@ci$ facter
architecture => i386
...
ipaddress => 10.0.2.15
ipaddress_eth0 => 10.0.2.15
ipaddress_eth1 => 192.168.33.16
```

```
...
uptime_seconds => 11365
virtual => virtualbox
vagrant@ci$ logout
```

Nesse caso, temos duas interfaces de rede configuradas: `10.0.2.15` na interface `eth0` corresponde ao endereço IP padrão alocado pelo Vagrant; `192.168.33.16` na interface `eth1` corresponde ao endereço IP que definimos no arquivo `Vagrantfile` para que os nossos servidores consigam se comunicar.

Podemos então reprovisionar o servidor `ci` rodando o comando `vagrant provision ci` e, se tudo der certo, você pode acessar a URL http://192.168.33.16/ no seu navegador e ver o repositório vazio mostrado na figura 7.1.

Figura 7.1: Repositório de pacotes vazio criado

Criando e publicando um pacote nativo

Com o repositório pronto, precisamos mudar nosso processo de *build* para criar um pacote nativo e publicá-lo. Usaremos a ferramenta **FPM** (https://github.com/jordansissel/fpm/) , uma Rubygem escrita por Jordan Sissel

para simplificar o processo de criar pacotes nativos para as diferentes plataformas – .deb no Debian/Ubuntu, .rpm no RedHat/CentOS etc.

Para instalar o FPM vamos adicionar um novo recurso do tipo Package no início da classe loja_virtual::ci, no arquivo modules/loja_virtual/manifests/ci.pp:

```
class loja_virtual::ci {
  include loja_virtual

  package { ['git', 'maven2', 'openjdk-6-jdk', 'rubygems']:
    ensure => "installed",
  }

  package { 'fpm':
    ensure   => 'installed',
    provider => 'gem',
    require  => Package['rubygems'],
  }
  ...
}
```

O recurso Package do Puppet irá usar o gerenciador de pacotes padrão do servidor em que está rodando, porém você pode sobrescrevê-lo. Nesse caso, estamos definindo o provedor gem explicitamente, para que o Puppet instale uma Rubygem e não um pacote nativo. Também precisamos instalar o pacote nativo rubygems, que é uma dependência do FPM.

Agora podemos alterar o arquivo de configuração do *build* da loja virtual para rodar um comando ao final de todo *build* bem sucedido. Eis as alterações necessárias no final do arquivo XML modules/loja_virtual/templates/config.xml:

```
<?xml version='1.0' encoding='UTF-8'?>
<maven2-moduleset plugin="maven-plugin@2.1">
  ...
  <prebuilders/>
  <postbuilders>
    <hudson.tasks.Shell>
      <command>
```

```
        fpm -s dir -t deb -C combined/target --prefix \
        /var/lib/tomcat7/webapps/ -d tomcat7 -n devopsnapratica
         \ -v $BUILD_NUMBER.master -a noarch devopsnapratica.war
         \ &amp;&amp; reprepro -V -b <%= repo_dir %> includedeb
         \ <%= repo_name %> *.deb
      </command>
    </hudson.tasks.Shell>
  </postbuilders>
  <runPostStepsIfResult>
    <name>SUCCESS</name>
    <ordinal>0</ordinal>
    <color>BLUE</color>
    <completeBuild>true</completeBuild>
  </runPostStepsIfResult>
</maven2-moduleset>
```

Vamos quebrar as alterações por partes, começando de trás para frente: o elemento `<runPostStepsIfResult>` foi alterado para dizer ao Jenkins que ele deve rodar um novo comando sempre que um *build* desse *job* completar e seu resultado for bem sucedido. O novo elemento `<hudson.tasks.Shell>` – dentro do elemento `<postbuilders>` – foi criado para dizer ao Jenkins que ele deve rodar um comando do terminal, ou *shell*.

O comando em si pode ser quebrado em duas partes: a primeira cria o pacote `.deb` usando o FPM e a segunda publica o pacote no repositório usando o `reprepro`. Vamos explicar cada opção passada ao comando `fpm`:

- `-s`: especifica que a origem é um conjunto de arquivos e/ou diretórios;
- `-t`: diz que o formato desejado é um pacote `.deb`;
- `-C`: diz ao FPM que ele precisa entrar no diretório `combined/target` para encontrar os arquivos que serão empacotados;
- `--prefix`: define o diretório alvo onde os arquivos devem ser colocados quando o pacote for instalado;

- `-d`: especifica uma dependência entre nosso pacote e o pacote `tomcat7`, uma vez que precisamos que o diretório das aplicações web exista antes do nosso pacote ser instalado;

- `-n`: o nome do nosso pacote `.deb`;

- `-v`: a versão do pacote. Nesse caso, estamos usando uma variável de ambiente definida pelo Jenkins `$BUILD_NUMBER` que corresponde ao número do *build* atual. Dessa maneira conseguimos que cada novo *build* produza uma nova versão do pacote;

- `-a`: especifica a arquitetura do pacote. No nosso caso não temos nenhum arquivo compilado para linguagem de máquina, então usamos o valor `noarch`;

- `devopsnapratica.war`: o último argumento passado é o nome do arquivo que queremos empacotar.

Após todas essas opções do comando `fpm`, usamos a sintaxe `&&` que é a forma de dizer `&&` dentro de um arquivo XML. O caractere `&` é especial dentro de um arquivo XML, por isso precisamos escapá-lo, usando o equivalente `&`. Depois disso invocamos o comando `reprepro` com as seguintes opções:

- `-V`: especifica que o comando deve ser mais verboso. Isso fará com que ele mostre mais mensagens explicativas a cada passo da sua execução;

- `-b`: especifica o diretório base onde o repositório está configurado. Nesse caso estamos usando o valor da variável `$repo_dir` que definimos no manifesto da classe `loja_virtual::ci`;

- `includedeb`: este é o comando do Reprepro para publicar um novo pacote no repositório. Este comando precisa de dois parâmetros: o nome do repositório, representado pela variável `$repo_name`, e o arquivo `.deb` desejado.

Com essa alteração no arquivo de configuração do *job*, precisamos reprovisionar o servidor `ci` novamente, usando o comando `vagrant provision ci`.

Ao final da execução do Puppet, você precisará iniciar um novo *build* manualmente para testar a geração do pacote. Você pode fazer isso clicando no link *"Build Now"* na página http://192.168.33.16:8080/job/loja-virtual-devops/.

Para nossa surpresa, o novo *build* falha com o erro mostrado na figura 7.2. Uma das mensagens de erro diz *"Error: gpgme created no signature!"*, que nos informa que o processo de publicação falhou quando o repositório tentou assinar o pacote usando a ferramenta GPG.

Figura 7.2: Build falhou ao tentar publicar o pacote

O **GPG** (http://www.gnupg.org/) – ou *GnuPG* – é uma ferramenta para comunicação segura. Originalmente escrito para possibilitar a troca de e-mails de forma segura, ele se baseia em um esquema de criptografia com chaves. Explicar esse tópico a fundo está além do escopo do livro, porém basta saber que o GPG permite gerar pares de chave, trocar e verificar chaves, criptografar e descriptografar documentos e autenticar documentos usando assinaturas digitais.

Repositórios de pacote usam o GPG para assinatura digital de pacotes, por isso vamos precisar configurar um par de chaves. Ao gerar um par de chaves, criamos uma **chave pública** – que pode ser distribuída para verificação de

assinaturas – e uma **chave privada** – que deve ser protegida e será usada pelo repositório para assinar os pacotes quando eles forem publicados.

O processo de criação de chaves usa o gerador de números aleatórios do sistema operacional, porém exige um certo nível de entropia no sistema para garantir a robustez do processo de criptografia. Entropia é gerada quando você usa dispositivos de entrada ao digitar no teclado, mover o *mouse*, ler arquivos do disco etc. Como estamos usando uma máquina virtual é mais difícil gerar entropia, por isso vamos instalar uma ferramenta chamada **haveged** (http://www.issihosts.com/haveged/) que nos ajudará no processo de geração de chaves:

```
$ vagrant ssh ci
vagrant@ci$ sudo apt-get install haveged
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  haveged
0 upgraded, 1 newly installed, 0 to remove and 157 not upgraded.
...
Setting up haveged (1.1-2) ...
```

Como o dono do repositório é o usuário `jenkins`, precisamos gerar as chaves GPG como este usuário. Para isso, vamos usar o comando `su` para trocar de usuário:

```
vagrant@ci$ sudo su - jenkins
jenkins@ci$ whoami
jenkins
```

Agora basta executar o comando `gpg --gen-key` para gerar o par de chaves. No processo de geração de chaves você precisará responder a algumas perguntas. A primeira delas é o tipo de chave, que usaremos a opção `(1) RSA and RSA`:

```
jenkins@ci$ gpg --gen-key
gpg (GnuPG) 1.4.11; Copyright (C) 2010 ...
```

```
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection?
```

A próxima opção é o tamanho da chave, que usaremos o padrão de 2048 bits:

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
```

Ao perguntar a validade da chave, usaremos a opção zero que diz que a chave não expira. Logo após essa opção, você precisa digitar `y` para confirmar que não quer que a chave expire:

```
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y
```

Agora você precisa especificar o nome e e-mail associados com esse par de chaves. Usaremos o nome *Loja Virtual* e o e-mail *admin@devopsnapratica.com.br*:

```
You need a user ID to identify your key; ...

Real name: Loja Virtual
Email address: admin@devopsnapratica.com.br
Comment:
You selected this USER-ID:
    "Loja Virtual <admin@devopsnapratica.com.br>"
```

Ao final de todas as opções, você precisa digitar a opção `O` para confirmar todos os dados. O GPG irá então pedir por um *passphrase*, um tipo de senha para proteger a chave. Nesse caso vamos deixar o *passphrase* em branco, porém isso não é recomendado em um ambiente de produção. O processo de assinatura será executado pelo *job* do Jenkins, então vamos confirmar o *passphrase* em branco:

```
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.
...
pub   2048R/4F3FD614 2014-03-06
      Key fingerprint =
        C3E1 45E1 367D CC8B 5863  FECE 2F1A B596 ...
uid                 Loja Virtual <admin@devopsnapratica.com.br>
sub   2048R/30CA4FA3 2014-03-06
```

No final da execução do comando você verá um resumo do par de chaves recém-criado. Em especial, você precisará anotar o ID da chave pública gerada, nesse caso o valor é `4F3FD614`. A última coisa que precisamos fazer é exportar a chave pública para um arquivo que ficará disponível na raiz do repositório:

```
jenkins@ci$ gpg --export --armor
  admin@devopsnapratica.com.br > \
  > /var/lib/apt/repo/devopspkgs.gpg
```

O comando `gpg --export` exporta o conteúdo da chave pública. A opção `--armor` indica qual par de chaves queremos exportar, nesse caso usando o e-mail `admin@devopsnapratica.com.br`. A saída deste comando é redirecionada para um novo arquivo `/var/lib/apt/repo/devopspkgs.gpg` que será exposto junto com o repositório. Feito isso, podemos deslogar da máquina virtual executando o comando `logout` duas vezes, uma para deslogar o usuário `jenkins` e outra para sair completamente da máquina virtual:

```
jenkins@ci$ logout
vagrant@ci$ logout
$
```

Agora podemos rodar um novo *build* clicando no link *"Build Now"* novamente. Dessa vez o *build* deve passar conforme mostra a figura 7.3.

Figura 7.3: Build passou e pacote foi publicado

Você pode verificar que o pacote foi publicado no repositório acessando a URL http://192.168.33.16/ que deverá ter algo parecido com a figura 7.4.

Figura 7.4: Repositório com o pacote publicado

Fazendo deploy do pacote nativo

Agora que temos um repositório de pacotes disponível com a última versão da loja virtual, podemos finalmente nos livrar o arquivo `.war` que está fixo dentro do módulo Puppet. Para isto, vamos substituir o recurso `File['/var/lib/tomcat7/webapps/devopsnapratica.war']` por dois novos recursos na classe `loja_virtual::web` no final do arquivo `modules/loja_virtual/manifests/web.pp`:

```
class loja_virtual::web {
  ...
  file { $loja_virtual::params::keystore_file: ... }
  class { "tomcat::server": ... }

  apt::source { 'devopsnapratica':
    location    => 'http://192.168.33.16/',
    release     => 'devopspkgs',
    repos       => 'main',
    key         => '4F3FD614',
    key_source  => 'http://192.168.33.16/devopspkgs.gpg',
    include_src => false,
```

```
  }

  package { "devopsnapratica":
    ensure => "latest",
    notify => Service["tomcat7"],
  }
}
```

O recurso `Apt::Source['devopsnapratica']` declara um novo repositório de pacotes que o APT pode usar. O parâmetro `location` possui o endereço do servidor `ci` onde o repositório está acessível por HTTP. O parâmetro `release` é o nome do repositório desejado, enquanto o parâmetro `repos` lista os componentes desejados. Os parâmetros `key` e `key_source` são configurações para verificação de assinatura então usamos o ID da chave pública que anotamos na seção anterior e a URL para acessar a chave pública no repositório. Por fim, o parâmetro `include_src` precisa ser configurado com o valor `false` pois nosso repositório não possui pacotes com o código-fonte do software distribuído.

Com o repositório configurado, criamos um novo recurso `package` para instalar a versão mais recente do pacote `devopsnapratica` e notificamos o serviço do Tomcat para que seja reiniciado após o pacote ser instalado. Feito isso, podemos reprovisionar o servidor `web` do zero, destruindo e reconstruindo sua máquina virtual:

```
$ vagrant destroy web
    web: Are you sure you want to destroy the 'web' VM? [y/N] y
==> web: Forcing shutdown of VM...
==> web: Destroying VM and associated drives...
==> web: Running cleanup tasks for 'puppet' provisioner...
$ vagrant up web
Bringing machine 'web' up with 'virtualbox' provider...
==> web: Importing base box 'hashicorp/precise32'...
==> web: Matching MAC address for NAT networking...
...
notice: Finished catalog run in 51.74 seconds
```

Se tudo der certo, você deve conseguir acessar a loja virtual na URL http://192.168.33.12:8080/devopsnapratica/ e pode então apagar o arquivo

`devopsnapratica.war` do diretório `modules/loja_virtual/files`.

7.2 Integração contínua do código de infraestrutura

Já temos uma forma de publicar e consumir pacotes com novas versões da loja virtual ao final de cada *build*. No entanto, todo código de infraestrutura que escrevemos até agora existe apenas na nossa máquina local junto com o arquivo de configuração do Vagrant. Não estamos aproveitando os benefícios da integração contínua no código Puppet.

Nesta seção vamos consertar isso, criando um novo repositório para controle de versões, um novo processo de *build*, adicionando testes automatizados e criando um novo *job* no Jenkins para executar esse novo processo de integração e publicar um novo pacote do código de infraestrutura a cada *commit*.

Criando o repositório para controle de versões

A primeira decisão é onde comitar o código de infraestrutura. Temos duas possibilidades: colocar junto com o repositório da aplicação ou criar um novo repositório para o código Puppet. A vantagem de colocar tudo no mesmo repositório é que você simplifica o processo de *setup* para um novo membro da sua equipe, pois tudo está no mesmo lugar. No entanto, você acopla mudanças na aplicação com mudanças na infraestrutura: um *commit* em qualquer dos dois componentes irá iniciar um processo de *build* que gera ambos os artefatos. Como o ciclo de mudanças da aplicação é relativamente independente da infraestrutura, vamos optar por usar repositórios separados.

Você pode optar por inicializar um novo repositório Git na raiz do diretório onde estão os módulos e manifestos do Puppet, usando o comando `git init`, alterando o arquivo `.gitignore` para indicar quais arquivos você não quer no repositório e fazendo o *commit* inicial:

```
$ git init
Initialized empty Git repository in /tmp/devops-puppet/.git/
$ echo ".vagrant" >> .gitignore
$ echo "librarian/.tmp" >> .gitignore
```

```
$ echo "librarian/.librarian" >> .gitignore
$ echo "librarian/modules" >> .gitignore
$ echo "librarian/Puppetfile.lock" >> .gitignore
$ echo "*.tgz" >> .gitignore
$ git add -A
$ git commit -m"Commit inicial"
[master (root-commit) 64686a5] Commit inicial
 27 files changed, 664 insertions(+)
 create mode 100644 .gitignore
 ...
 create mode 100644 modules/tomcat/templates/server.xml
```

Depois você pode criar um novo repositório no GitHub manualmente escolhendo um nome, por exemplo `loja-virtual-puppet`. Feito isso, você pode executar o comando a seguir substituindo o usuário `dtsato` pelo seu usuário do GitHub e `loja-virtual-puppet` pelo nome do repositório que você escolheu:

```
$ git remote add origin
                  git@github.com:dtsato/loja-virtual-puppet.git
$ git push -u origin master
Counting objects: 42, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (33/33), done.
Writing objects: 100% (42/42), 12.11 KiB | 0 bytes/s, done.
Total 42 (delta 1), reused 0 (delta 0)
To git@github.com:dtsato/loja-virtual-puppet.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

A outra opção, que vai economizar alguns passos nas próximas seções, é clonar o repositório já existente no repositório https://github.com/dtsato/loja-virtual-devops-puppet. Da mesma forma que fizemos no capítulo 6, você pode clonar esse repositório clicando no botão *"Fork"*.

Independente da sua escolha, temos agora nosso código Puppet em um sistema de controle de versões e podemos seguir adiante para criar um *build* automatizado.

Um build automatizado simples

O Puppet é escrito em Ruby e diversas das ferramentas usadas pela comunidade são inspiradas ou integradas com o ecossistema Ruby. Ruby possui uma ferramenta de *build* bastante poderosa e ao mesmo tempo simples: o **Rake** (http://rake.rubyforge.org/) . Inspirado pelo Make, você pode definir **tarefas** (ou *tasks*) com pré-requisitos que definem comandos arbitrários.

Para entender como o Rake funciona, vamos criar um arquivo `Rakefile` onde podemos definir nossas primeiras tarefas:

```
desc "Cria o pacote puppet.tgz"
task :package do
  sh "tar czvf puppet.tgz manifests modules librarian/modules"
end

desc "Limpa o pacote puppet.tgz"
task :clean do
  sh "rm puppet.tgz"
end
```

A tarefa `package` cria um pacote usando o comando `tar` contendo todo o código Puppet que temos até agora. A tarefa `clean` apaga o pacote gerado usando o comando `rm`. Para executar essas tarefas, você pode simplesmente passá-las para o comando `rake` no terminal:

```
$ rake -T
rake clean    # Limpa o pacote puppet.tgz
rake package  # Cria o pacote puppet.tgz
$ rake package
tar czvf puppet.tgz manifests modules librarian/modules
a manifests
...
a librarian/modules/apache/files/httpd
$ rake clean
rm puppet.tgz
```

Também veja que o comando `rake -T` lista todas as tarefas documentadas. Sabemos que os módulos do Librarian Puppet já foram baixados, por isso eles entraram no pacote. Porém eles não estariam disponíveis se o Librarian

Puppet não tivesse executado. Podemos criar uma outra tarefa para executar o comando `librarian-puppet install` e ao mesmo tempo aprender a sintaxe do Rake para especificar pré-requisitos entre tarefas:

```
# encoding: utf-8
namespace :librarian do
  desc "Instala os módulos usando o Librarian Puppet"
  task :install do
    Dir.chdir('librarian') do
      sh "librarian-puppet install"
    end
  end
end

desc "Cria o pacote puppet.tgz"
task :package => 'librarian:install' do
  sh "tar czvf puppet.tgz manifests modules librarian/modules"
end
...
```

O Rake também suporta ***namespaces***, para agrupar tarefas similares dentro de um mesmo nome. Nesse caso, criamos um *namespace* chamado `librarian` e uma tarefa `install` dentro dele que simplesmente entra no diretório `librarian` e executa o comando `librarian-puppet install`. A definição da tarefa `package` também mudou para declarar sua dependência com a nova tarefa `librarian:install`. Ao executar a tarefa `package` novamente, vemos um erro:

```
$ rake package
librarian-puppet install
rake aborted!
Command failed with status (127): [librarian-puppet install...]
...
Tasks: TOP => package => librarian:install
(See full trace by running task with --trace)
```

Isso acontece pois o Librarian Puppet não está instalado no seu sistema. Nós apenas o usamos como um plugin do Vagrant. No mundo Ruby, a melhor forma de gerenciar esse tipo de dependência de ambiente é usar o **Bundler**

(http://bundler.io/) . Com o Bundler você declara suas dependências em um arquivo `Gemfile` e o Bundler toma conta de resolver as dependências, baixar e instalar as Rubygems nas versões que você desejar. Vamos criar o arquivo `Gemfile` com o seguinte conteúdo:

```
source 'https://rubygems.org'

gem 'rake', '10.1.1'
gem 'puppet-lint', '0.3.2'
gem 'rspec', '2.14.1'
gem 'rspec-puppet', '1.0.1'
gem 'puppet', '2.7.19'
gem 'librarian-puppet', '0.9.14'
```

Além do Rake e do Librarian Puppet, já adicionamos algumas outras dependências que vamos precisar na próxima seção. Perceba também que estamos usando versões específicas dessas Rubygems para tornar nosso ambiente local o mais próximo possível do ambiente de produção. Por exemplo, nossas máquinas virtuais rodam o Puppet na versão `2.7.19`, por isso usamos a mesma versão no processo de *build*.

Para instalar as dependências você precisa primeiramente instalar o Bundler em si, caso nunca o tenha usado na sua máquina. Feito isso, você pode rodar o comando `bundle install` para instalar todas as Rubygems necessárias:

```
$ gem install bundler
Successfully installed bundler-1.5.3
1 gem installed
$ bundle install
Fetching gem metadata from https://rubygems.org/...........
Fetching additional metadata from https://rubygems.org/..
Resolving dependencies...
Using rake (10.1.1)
...
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is
installed.
```

Agora a tarefa `package` deve rodar com sucesso. Só precisamos prefixar o comando com `bundle exec` para garantir que o Rake vai rodar usando as gems nas versões especificadas no nosso `Gemfile`:

```
$ bundle exec rake package
librarian-puppet install
tar czvf puppet.tgz manifests modules librarian/modules
a manifests
a manifests/ci.pp
...
```

Temos um processo automatizado para empacotar nosso código, porém não estamos fazendo nenhum tipo de verificação. Uma forma bem simples e rápida de validar o código Puppet é usar a ferramenta **Puppet Lint** (http://puppet-lint.com/) , que verifica que seu código está seguindo um padrão de código e gera avisos quando encontrar problemas. Como nós já o incluímos no nosso `Gemfile`, podemos simplesmente adicionar uma nova tarefa e dependência no arquivo `Rakefile`:

```
# encoding: utf-8
require 'puppet-lint/tasks/puppet-lint'

PuppetLint.configuration.ignore_paths = ["librarian/**/*.pp"]

namespace :librarian do ... end

desc "Cria o pacote puppet.tgz"
task :package => ['librarian:install', :lint] do
  sh "tar czvf puppet.tgz manifests modules librarian/modules"
end

task :clean do ... end
```

O Puppet Lint já define a tarefa Rake para gente, por isso nós precisamos apenas fazer um `require` do arquivo `'puppet-lint/tasks/puppet-lint'`. Também configuramos a tarefa para ignorar os arquivos `.pp` dentro do diretório `librarian` pois não queremos validar o código Puppet dos módulos externos que estamos

usando. Nosso *build* deve focar nos módulos que estamos escrevendo. Por fim, alteramos os pré-requisitos da tarefa `package` para listar duas tarefas usando um `Array`.

Ao executar a tarefa `package`, veremos diversos avisos do Puppet Lint, representados como mensagens do tipo `WARNING`:

```
$ bundle exec rake package
librarian-puppet install
modules/loja_virtual/manifests/ci.pp - \
WARNING: class not documented on line 1
modules/loja_virtual/manifests/ci.pp - \
WARNING: double quoted string containing no variables on line 5
...
```

Testando código Puppet

O Puppet Lint vai encontrar problemas de sintaxe e estilo no código Puppet, porém não consegue verificar se você digitou o nome de uma variável errado ou se um determinado valor está errado. Para isso, podemos escrever testes automatizados para nossos módulos usando ferramentas como o **RSpec-Puppet** (http://rspec-puppet.com/) , uma extensão do popular *framework* de testes RSpec que nos dá diversas extensões para testar código Puppet com mais facilidade.

A Rubygem do RSpec-Puppet também já está instalada, porém precisamos inicializar uma estrutura de diretórios onde colocaremos nosso código de teste dentro de cada módulo. Primeiramente, iremos focar no módulo `mysql`:

```
$ cd modules/mysql
$ bundle exec rspec-puppet-init
 + spec/
 ...
 + Rakefile
$ rm Rakefile
$ cd ../..
```

Nosso primeiro teste será para a classe `mysql::client`, por isso vamos criar um novo arquivo `client_spec.rb` dentro do diretório

modules/mysql/spec/classes com o seguinte conteúdo:

```
require File.join(File.dirname(__FILE__), '..', 'spec_helper')

describe 'mysql::client' do
  it {
    should contain_package('mysql-client').
      with_ensure('installed')
  }
end
```

O bloco describe define o nome da classe, tipo definido ou função do nosso módulo que queremos testar. Cada bloco it representa um caso de teste. Neste caso, a classe mysql::client instala o pacote mysql-client por isso nosso teste verifica que o pacote foi instalado. Para executar o teste, podemos usar o comando rspec:

```
$ bundle exec rspec modules/mysql/spec/classes/client_spec.rb
.

Finished in 0.34306 seconds
1 example, 0 failures
```

Sucesso! Temos nosso primeiro teste do Puppet passando. Assim como no capítulo 6, nosso objetivo aqui não é obter uma alta cobertura de testes, mas sim entender como integrar o processo de teste com o processo de entrega contínua. Para mais detalhes de como escrever testes para código Puppet, você pode explorar o repositório original ou ler a documentação do RSpec-Puppet em http://rspec-puppet.com/tutorial/.

O que precisamos fazer é rodar os testes no processo de *build*. Para isso, vamos alterar o arquivo Rakefile para criar novas tarefas:

```
...
require 'rspec/core/rake_task'
TESTED_MODULES = %w(mysql)
namespace :spec do
  TESTED_MODULES.each do |module_name|
    desc "Roda os testes do módulo #{module_name}"
```

```
    RSpec::Core::RakeTask.new(module_name) do |t|
      t.pattern = "modules/#{module_name}/spec/**/*_spec.rb"
    end
  end
end

desc "Roda todos os testes"
task :spec => TESTED_MODULES.map {|m| "spec:#{m}" }

namespace :librarian do ... end

desc "Cria o pacote puppet.tgz"
task :package => ['librarian:install', :lint, :spec] do
  sh "tar czvf puppet.tgz manifests modules librarian/modules"
end

task :clean do ... end

task :default => [:lint, :spec]
```

Vamos usar a tarefa Rake fornecida pelo próprio RSpec para definir uma tarefa para cada módulo. Por enquanto, temos testes definidos apenas para o módulo `mysql`. Para rodar todos os testes, criamos uma tarefa `spec` que depende das tarefas de cada módulo. Também adicionamos um novo pré-requisito na tarefa `package` e definimos uma tarefa padrão `default` que executa sempre que você rodar o comando `rake` sem especificar nenhuma tarefa. Por exemplo:

```
$ bundle exec rake
modules/loja_virtual/manifests/ci.pp - \
WARNING: class not documented on line 1
...
/Users/dtsato/.rvm/rubies/ruby-1.9.3-p385/bin/ruby \
-S rspec modules/mysql/spec/classes/client_spec.rb
.

Finished in 0.33303 seconds
1 example, 0 failures
```

Com isso temos um processo de *build* e testes automatizado que podemos usar para implementar integração contínua do código de infraestrutura. Podemos então fazer um *commit* e enviar nosso código para o repositório Git:

```
$ git add -A
$ git commit -m"Adicionando build e testes automatizados"
[master 3cf891c] Adicionando build e testes automatizados
 8 files changed, 100 insertions(+)
 create mode 100644 Gemfile
 ...
 create mode 100644 modules/mysql/spec/spec_helper.rb
$ git push
Counting objects: 21, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (18/18), 2.34 KiB | 0 bytes/s, done.
Total 18 (delta 0), reused 0 (delta 0)
To git@github.com:dtsato/loja-virtual-puppet.git
   fe7a78b..3cf891c  master -> master
```

Configurando um job no Jenkins para o código Puppet

Agora podemos finalmente criar um novo *job* no Jenkins para nosso código Puppet. Acessando a página principal do Jenkins em http://192.168.33.16:8080/ você pode clicar no link *New Item*. Na próxima página, especificamos o nome do *job* como *loja-virtual-puppet* e o tipo de projeto é *Build a free-style software project* conforme mostra a figura 7.5.
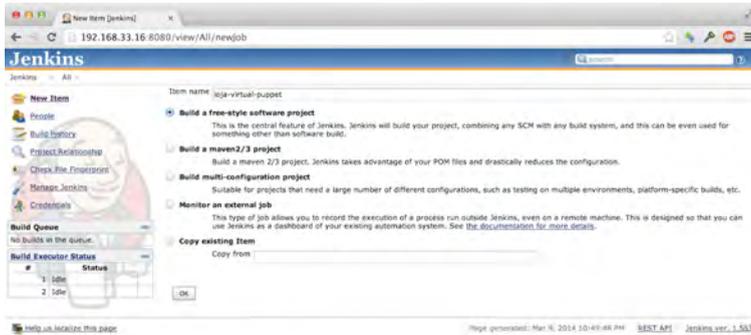
Figura 7.5: Novo job para código de infraestrutura

Ao clicar no botão *OK*, chegamos na tela principal de configuração do novo *job*. Vamos descrever passo a passo o que precisamos fazer em cada seção:

- **Seção *"Source Code Management"***: vamos escolher a opção *Git* e preencher o campo *Repository URL* com a URL do seu repositório no GitHub criado na seção anterior, conforme mostra a figura 7.6.
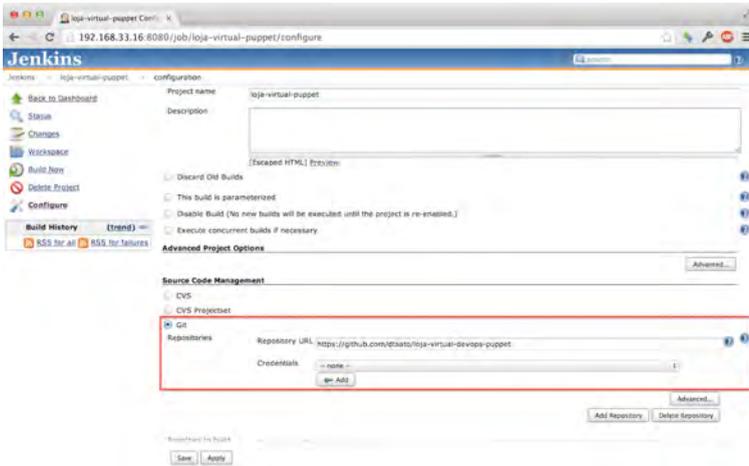
Figura 7.6: Configurando o sistema de controle de versões

- **Seção *"Build Triggers"***: vamos escolher a opção *Poll SCM* e preencher a caixa de texto *Schedule* com o valor * * * * *, conforme mostra a figura 7.7
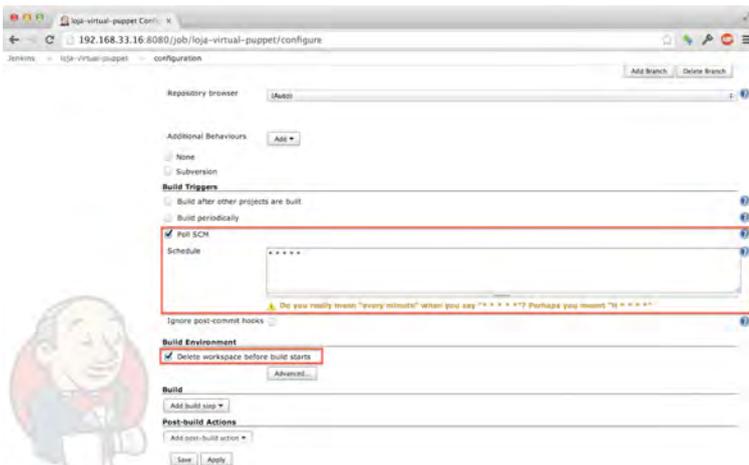
Figura 7.7: Configurando gatilho e limpeza da área de trabalho

- **Seção *"Build Environment"***: nesta seção vamos escolher a opção *Delete workspace before build starts* para limpar a área de trabalho antes de começar um novo *build*, conforme mostra a figura 7.7.
- **Seção *"Build"***: nesta seção escolheremos a opção *Execute shell* dentro da caixa de seleção *Add build step*. O campo *Command* deverá ser preenchido com o valor `bundle install --path ~/.bundle && bundle exec rake package`, conforme mostra a figura 7.8.
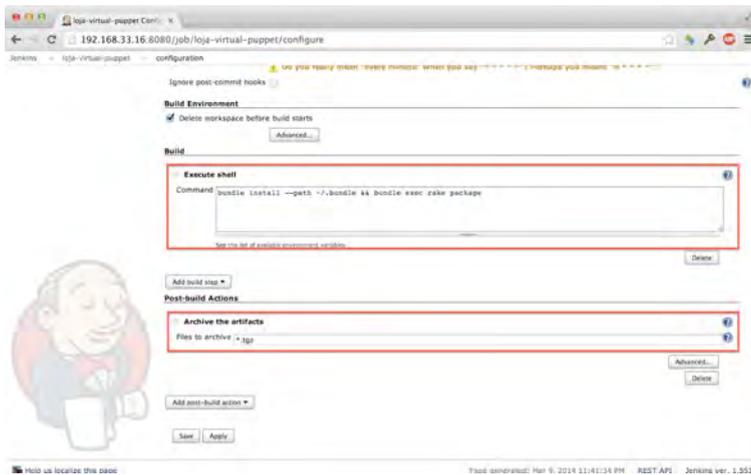
Figura 7.8: Comando do build e arquivando artefatos no final do build

- **Seção *"Post-build Actions"***: ao clicar no botão *Add post-build action* vamos escolher a opção *Archive the artifacts* e preencher o campo *Files to archive* com o valor **.tgz*, conforme mostra a figura 7.8.

Com isso podemos clicar no botão *Save* e um novo *build* deverá ser agendado e executar nos próximos minutos. No entanto, o *build* falha com uma mensagem *bundle: not found*. Vamos precisar instalar o Bundler para que o Jenkins consiga rodar nosso *build*. Faremos isso adicionando um novo recurso do tipo `Package` no começo do arquivo `modules/loja_virtual/manifests/ci.pp`, junto com a instalação do FPM:

```
class loja_virtual::ci {
  ...
  package { ['fpm', 'bundler']:
    ensure   => 'installed',
    provider => 'gem',
    require  => Package['rubygems'],
  }
  ...
}
```

Após reprovisionarmos o servidor `ci` com o comando `vagrant provision ci`, você pode iniciar um novo *build* clicando no link *"Build Now"*. Dessa vez ele deve terminar com sucesso, conforme mostra a figura 7.9.
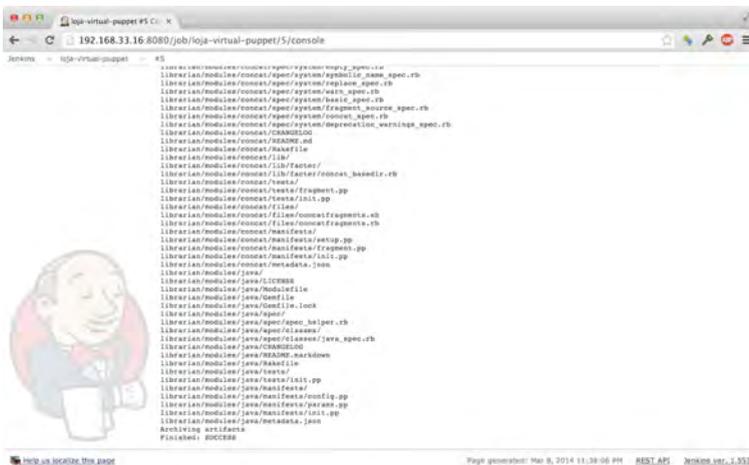
Figura 7.9: Build bem-sucedido

Seguindo os mesmos passos do capítulo 6 podemos automatizar o processo de criação desse *job* com código Puppet. Encorajamos que você tente fazê-lo para exercitar seu conhecimento do Puppet e dos conceitos apresentados até agora. Uma possível solução está disponível no repositório do GitHub que você clonou no início desta seção. Iremos omitir essa explicação do livro

por questão de espaço e para podermos discutir um dos conceitos mais importantes da entrega contínua: a **pipeline de entrega**.

7.3 Pipeline de entrega

Para terminar este capítulo, é hora de integrar nossos dois *builds* e configurar um processo de *deploy* para produção em um clique. Atualmente temos dois processos de integração contínua independentes, nos quais o produto final de cada *build* é um pacote que pode ir para produção.

Geralmente o caminho de um *commit* até a produção precisa passar por diversas etapas, algumas delas são automatizadas enquanto outras exigem aprovações manuais. Por exemplo, é comum empresas terem etapas de validação com o usuário onde você pode demonstrar o software funcionando em um ambiente de testes e só após um consenso mútuo é que o código vai para produção. Existem também diversas empresas web que colocam qualquer *commit* bem sucedido em produção sem nenhuma etapa de validação manual.

É aqui que fazemos a distinção entre duas práticas que geralmente são confundidas: **implantação contínua** (*deploy* contínuo ou *Continuous Deployment*) é a prática de colocar em produção todo *commit* bem sucedido e geralmente significa fazer diversos *deploys* em produção por dia; **entrega contínua** é a prática onde qualquer *commit* pode ir para produção a qualquer momento, porém a decisão de quando isto acontece é uma opção de negócio.

Para exemplificar a diferença vamos pensar em duas situações: na primeira você está desenvolvendo um produto que precisa ser distribuído e instalado – por exemplo, um aplicativo móvel – e fazer um *deploy* em produção a cada *commit* não é algo fácil ou desejável. Em outra situação você tem uma aplicação web usada constantemente por vários usuários e colocar correções ou novas funcionalidades no ar rapidamente podem te dar uma vantagem competitiva no mercado.

Independente de qual prática fizer mais sentido para você, existe um padrão essencial para implementar entrega ou implantação contínua: a **pipeline de entrega**. Em um nível abstrato, a pipeline de entrega é uma manifestação automatizada do seu processo de entrega de software, desde uma mudança até

sua implantação em produção. Ela modela as etapas automatizadas e manuais do processo de entrega e é uma extensão natural da prática de integração contínua que discutimos até aqui.

A figura 7.10 mostra um exemplo de uma pipeline de entrega comum, onde uma mudança passa pelos estágios de *build* e testes de unidade, testes de aceitação, validação com usuário final e entrega em produção. O diagrama mostra também como a pipeline fornece feedback rápido quando problemas são encontrados. Cada estágio está lá para vetar a mudança antes que ela chegue em produção. Somente os *commits* que "sobreviverem" todos os estágios da pipeline são disponíveis para *deploy* em produção.
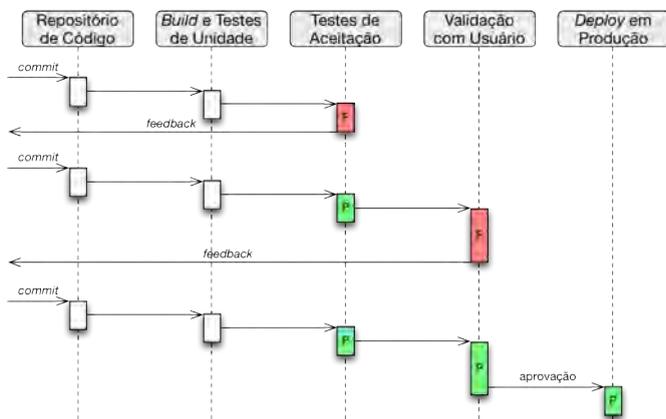
Figura 7.10: Mudanças se propagando pela pipeline de entrega

Modelar a sua pipeline de entrega exige entender seu processo de entrega e balancear o nível de feedback que você quer receber. Geralmente, os estágios iniciais fornecem feedback mais rápido, porém podem não estar executando testes exaustivos em um ambiente parecido com produção. Conforme os estágios avançam, a sua confiança aumenta, assim como a similaridade dos ambientes que vão ficando cada vez mais parecidos com produção.

No nosso caso, vamos modelar uma pipeline de entrega bem simplificada, uma vez que nosso objetivo é mostrar uma implementação prática. Por exemplo, nossos testes de aceitação atualmente rodam no *build* inicial da aplicação,

junto com os testes de unidade, pois é rápido o suficiente rodá-los em conjunto. Conforme a cobertura dos testes funcionais aumenta, geralmente aumenta também seu tempo de execução e você vai precisar considerar separá-los em estágios diferentes da pipeline.

A figura 7.11 mostra a nossa pipeline de entrega. Temos dois estágios iniciais e paralelos para *build*, teste e empacotamento do código da aplicação e do código de infraestrutura. Ambos os estágios estão conectados com um novo estágio de *deploy* para produção que iremos configurar a seguir. Por enquanto, todas as transições serão automáticas portanto essa pipeline implementa um processo de implantação contínua.
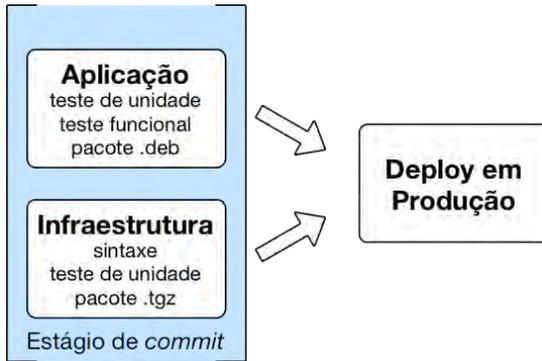
Figura 7.11: Pipeline de entrega da loja virtual

Criando um novo estágio de deploy para produção

Antes de criar o novo *job* no Jenkins, precisamos instalar mais dois plugins que permitem ligar nossos *jobs* entre si: o plugin *"Parameterized Trigger"* (https://wiki.jenkins-ci.org/display/JENKINS/Parameterized+Trigger+Plugin) permite que um *build* seja engatilhado quando o *build* de outro projeto termina, assim como permite passar parâmetros entre eles; o plugin *"Copy Artifact"* (https://wiki.jenkins-ci.org/display/JENKINS/Copy+Artifact+Plugin) permite copiar artefatos de um *build* para outro.

Vamos adicionar esses plugins na classe `loja_virtual::ci`, alterando o arquivo `modules/loja_virtual/manifests/ci.pp`:

```
class loja_virtual::ci {
  ...
  $plugins = [
    ...
    'ws-cleanup',
    'parameterized-trigger',
    'copyartifact'
  ]
  ...
}
```

Com isso podemos reprovisionar o servidor `ci` executando o comando `vagrant provision ci` e estaremos pronto para criar o novo *job* no Jenkins. Na página principal do Jenkins criaremos um novo *job* clicando no link *New Item* e preenchendo o nome do projeto como *Deploy em Produção* e escolhendo o tipo de projeto *Build a free-style software project*.

Ao clicar *"OK"* seremos levados para a página de configuração do nosso novo *job*. Dessa vez deixaremos a seção *"Source Code Management"* intocada pois queremos que nosso *build* rode quando os outros projetos terminarem. Para isso, na seção *Build Triggers* vamos escolher a opção *Build after other projects are built* e preencher o campo *Project names* com o valor *loja-virtual-devops, loja-virtual-puppet*.

Na seção *Build Environment* vamos novamente marcar a opção *Delete workspace before build starts*. Na seção *Build* iremos adicionar dois comandos:

- **Copiando artefato do Puppet**: selecionando o tipo *Copy artifacts from another project*, vamos preencher o campo *Project name* com o valor *loja-virtual-puppet* e escolheremos a opção *Upstream build that triggered this job*. Isso fará com que o Jenkins copie o artefato do *build* que engatilhou essa execução. Além disso, vamos marcar a opção *Use Last successul build as fallback* para que possamos engatilhar um *deploy* para produção manualmente sem precisar rodar o *build* do projeto anterior. Por fim, preenchemos o campo *Artifacts to copy* com o valor *puppet.tgz*, conforme mostra a figura 7.12.

- **Fazendo o *deploy* no servidor web**: adicionamos outro comando do tipo *Execute shell*, vamos preencher o campo *"Command"* com um

comando de três partes, separadas por `&&`. A primeira parte copia o artefato para o servidor `web` usando o `scp`. A segunda parte executa um comando remoto no servidor `web` para descompactar o artefato copiado. A terceira parte irá iniciar a execução do Puppet usando o comando `puppet apply` e colocando nossos módulos no `modulepath`. O comando completo, que pode ser visto na figura 7.12, fica:

```
scp puppet.tgz vagrant@192.168.33.12: && \
ssh vagrant@192.168.33.12 'tar zxvf puppet.tgz' && \
ssh vagrant@192.168.33.12 'sudo /opt/vagrant_ruby/bin/puppet
apply \ --modulepath=modules:librarian/modules manifests/web.pp'
```
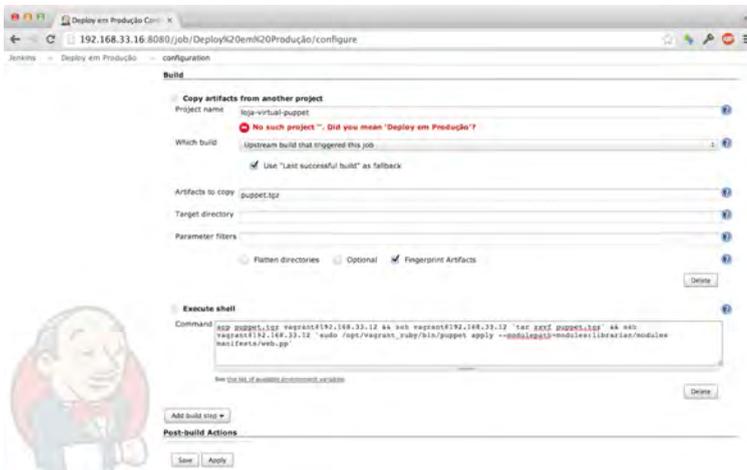
Figura 7.12: Comandos para deploy da loja virtual

Ao clicar em *Save* temos o nosso novo *job* para fazer *deploy* em produção. Vamos iniciar o *deploy* clicando no botão *Build Now* no novo projeto. Para nossa surpresa, o *deploy* falha com um erro *Permission denied, please try again.* conforme mostra a figura 7.13.

Figura 7.13: Erro na tentativa de deploy automatizado

Isto acontece pois o usuário `jenkins` que executa todos os *builds* não tem permissão para copiar ou executar comandos no servidor `web`. Para consertar isso, precisamos logar na máquina virtual e executar alguns comandos para troca de chaves SSH entre os servidores:

```
$ vagrant ssh ci
vagrant@ci$ sudo su - jenkins
```

O comando `su` muda a sessão para o usuário `jenkins`. Feito isso, podemos rodar o comando `ssh-keygen` que irá gerar um par de chaves SSH e fará algumas perguntas:

```
jenkins@ci$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/var/lib/jenkins/.ssh/
                                                        id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /var/lib/jenkins/.ssh/
                                                        id_rsa.
Your public key has been saved in /var/lib/jenkins/.ssh/
```

```
                                                    id_rsa.pub.
The key fingerprint is:
67:ae:7d:37:ec:de:1f:e7:c3:a8:15:50:49:02:9d:49
jenkins@precise32
The key's randomart image is:
+--[ RSA 2048]----+
|         .+E+o.  |
|           +o.   |
|            .    |
|             .   |
|        S o   .  |
|         +     . |
|          .  oo..|
|         o  o.+=o|
|        .  .o.+o.*|
+-----------------+
```

Usamos a resposta padrão para todas as perguntas, inclusive deixando o *passphrase* em branco. Com o par de chaves SSH criado, precisamos copiar nossa chave pública para o servidor `web`. Usaremos o comando `ssh-copy-id` para isso e a senha do usuário `vagrant` é também `vagrant`:

```
jenkins@ci$ ssh-copy-id vagrant@192.168.33.12
vagrant@192.168.33.12's password:
Now try logging into the machine, with ...
```

Com isso terminamos a configuração SSH e podemos sair de ambas as máquinas virtuais, novamente executando o comando `logout` duas vezes:

```
jenkins@ci$ logout
vagrant@ci$ logout
$
```

Feito isso, podemos iniciar um novo *deploy* para produção e dessa vez tudo funciona! A figura 7.14 mostra a saída bem sucedida do processo de *deploy*.

Figura 7.14: Deploy ao clique de um botão bem sucedido

7.4 Próximos Passos

Agora temos uma pipeline de entrega automatizada que nos permite fazer *deploy* automatizado a cada *commit* bem sucedido. Conforme discutimos durante o capítulo, existem mais algumas coisas que você pode explorar para fortalecer o seu entendimento dos conceitos apresentados. Você pode melhorar a cobertura dos testes do Puppet.

Outro exercício interessante é automatizar as configurações do Jenkins que fizemos manualmente neste capítulo. As técnicas que usamos no capítulo 6 podem ajudá-lo aqui. Ainda no tópico de infraestrutura como código, a classe `loja_virtual::ci` está começando a ficar complexa e após a automação da configuração dos novos *jobs*, é bem capaz que você precise refatorá-la.

Quando copiamos o arquivo `.tar` para o servidor `web`, deixamos os arquivos descompactados soltos por lá. O ideal seria fazer uma limpeza assim que o processo de *deploy* terminar para garantir que não deixamos nada para trás. Caso algum módulo seja removido em um *build* subsequente, não queremos ter problemas no processo de *deploy* por ter deixado arquivos soltos em um *deploy* anterior.

Nossa decisão de empacotar o código Puppet como um arquivo `.tar` e copiá-lo de um *build* para o outro usando um plugin do Jenkins é apenas uma das possíveis soluções. Existem outras opções de distribuição de código Puppet que não discutimos no livro, mas nada impede você de explorá-las. O Puppet possui um componente chamado de Puppet Master que é um servidor central onde os nós da sua infraestrutura podem se registrar para se manter atualizados. Como isso requer mais infraestrutura, nós decidimos seguir uma abordagem sem o Puppet Master e usar um pouco mais de orquestração externa.

Por fim, nosso processo de *deploy* apenas atualiza o servidor `web`. Gostaríamos de fazer um *deploy* completo de toda nossa infraestrutura, ou pelo menos dos servidores que afetam diretamente o ambiente de execução da aplicação. Um outro exercício interessante é pensar em como implementar esse *deploy* do servidor de banco de dados juntamente com o servidor `web`.

Capítulo 8

Tópicos avançados

Já discutimos diversos conceitos importantes para adotação de DevOps e implementamos práticas que possibilitam a entrega contínua. Além da colaboração entre desenvolvedores e operações, um dos aspectos culturais mais importantes de DevOps é a melhoria contínua. Assim como as reuniões de retrospectiva ajudam equipes ágeis a adaptar seu processo de desenvolvimento, precisamos usar técnicas semelhantes para avaliar e evoluir o processo de entrega de software.

Melhoria contínua é uma forma de aplicar o método científico no nosso trabalho. Você começa com uma hipótese de melhoria, com base no seu instinto ou em observações e fatos sobre o seu processo atual. Você então propõe um experimento para testar sua hipótese. Isso implica na coleta de novos dados, fatos e observações que podem suportar ou refutar sua hipótese inicial. Esse ciclo de aprendizado possibilita implementar pequenas alterações no seu processo para torná-lo mais e mais eficiente.

Neste capítulo iremos abordar alguns tópicos mais avançados. Nosso objetivo não é criar uma lista compreensiva de todas as técnicas, ferramentas, plataformas e metodologias que você precisa conhecer. Queremos abordar alguns tópicos importantes para que você tenha conhecimento suficiente para decidir quais são os próximos passos a partir daqui na sua jornada de melhoria contínua.

Evoluindo um processo de deploy manual para automatizado

No final de 2012, eu estava trabalhando com um cliente que fazia *deploy* em produção uma ou duas vezes por semana. Eles tinham um processo de *deploy* manual que precisava ser executado e acompanhado de perto por um dos engenheiros que ficava no serviço até mais tarde. Os testadores também ficavam trabalhando até tarde para realizar alguns testes manuais básicos logo após cada *deploy*.

Certo dia, esse engenheiro cometeu um erro e esqueceu de mudar um arquivo de configuração para usar a URL de conexão com o banco de dados corretas. Com isso, um dos processos de indexação estava usando o banco de dados errado. O processo não falhou e os testadores não perceberam o erro pois a busca ainda estava funcionando, porém com resultados incorretos. Por isso, o problema só foi detectado no dia seguinte quando um representante do negócio estava apresentando o produto para um potencial cliente.

Como esta falha teve um custo alto para o negócio, eu facilitei uma reunião de *post-mortem* com os envolvidos e fizemos uma **análise de causa raiz** discutindo o motivo de o problema ter acontecido assim como potenciais medidas para evitar que ele se repetisse. Um dos resultados da reunião foi que eu pareasse com o engenheiro no próximo *deploy* para levantar dados sobre os processos manuais que poderiam ser automatizados.

Na semana seguinte, fiquei no trabalho até mais tarde para acompanhar o processo de *deploy* e comecei a preencher *post-its* para cada passo manual que ele executava. Eu também tomava nota quando ele não tinha certeza do que fazer a seguir: quais comandos ele executava para decidir se podia seguir adiante ou não. Ao final daquela noite, tínhamos uma parede inteira coberta de *post-its*.

No dia seguinte, fizemos uma reunião com o resto da equipe e, usando os

post-its como dados, criamos várias ideias para automatizar as tarefas e verificações mais problemáticas. Não implementamos todas as sugestões imediatamente, mas usamos esse novo conhecimento para convencer os representantes do negócio para alocar uma certa capacidade da equipe para trabalhar nessas melhorias durante as próximas iterações.

Após alguns meses, o processo de *deploy* estava quase totalmente automatizado e descobrimos diversas tarefas manuais que poderiam ser executadas e testadas em outros ambientes antes de executá-las em produção. Apesar do processo automatizado, meu cliente ainda não estava confortável em executar *deploys* durante o dia, por isso ainda ficávamos no trabalho até tarde. No entanto agora tínhamos bem menos surpresas pois era só apertar um botão e esperar o processo terminar.

8.1 Fazendo deploy na nuvem

O custo de manter um *data center* com hardware físico é alto para empresas de pequeno e médio porte. Além disso, fazer planejamento de capacidade nessa situação é difícil pois você precisa prever sua taxa de crescimento antes mesmo de ter seu primeiro usuário.

Nos últimos anos, empresas começaram a investir em tecnologias de virtualização para tentar aproveitar melhor a capacidade dos servidores físicos que já possuíam. Com máquinas virtuais, você pode rodar mais de um servidor na mesma máquina física, ajudando a controlar os custos com compra e manutenção do hardware e do *data center*.

Enquanto isso, empresas da internet como a Amazon investiram muito dinheiro para montar uma infraestrutura robusta que aguentasse o pico de acessos e compras do final do ano. No restante do ano, os servidores são subutilizados, deixando recursos computacionais disponíveis. Foi daí que tiveram a ideia de alugar esse excesso de recursos para potenciais clientes, criando uma das inovações mais importantes no mundo da tecnologia dos últimos tempos: a computação em nuvem.

O modelo de **computação em nuvem** possibilita novas maneiras de contratar e pagar pela infraestrutura, apresentando cinco características essenciais [8]:

- **Serviço sob demanda**: um consumidor pode provisionar infraestrutura de computação sob demanda e de forma automatizada, sem a necessidade de interação humana com alguém do seu provedor de serviços.

- **Amplo acesso à rede**: os recursos da nuvem estão disponíveis na rede e podem ser acessados por clientes de qualquer tipo de plataforma – dispositivos móveis, *tablets*, *laptops* ou outros servidores.

- ***Pool* de recursos**: os recursos de computação do provedor fazem parte de um *pool* compartilhado que pode ser usado por vários consumidores. Geralmente, recursos na nuvem abstraem sua real localização dos consumidores. Em alguns casos, consumidores podem especificar a localização desejada, mas isso acontece em um nível de abstração mais alto – como um certo país, estado ou *data center*.

- **Elasticidade**: recursos podem ser provisionados e liberados rapidamente – até de forma automatizada em alguns casos – para que a infraestrutura possa escalar de acordo com a demanda. Do ponto de vista do consumidor, os recursos na nuvem são praticamente infinitos.

- **Serviço mensurado**: serviços na nuvem controlam e mensuram automaticamente o uso de cada recurso. Da mesma forma que um hidrômetro mede o quanto de água você gastou no mês, seu provedor irá usar esses dados para cobrar pelo uso. O uso desses recursos pode ser monitorado, controlado e gerar relatórios que trazem transparência tanto para o provedor quanto para o consumidor.

Existem também três modelos de serviço na computação em nuvem, com diferentes níveis de abstração:

- **IaaS** (*Infrastructure as a Service* ou Infraestrutura como Serviço): disponibiliza recursos de infraestrutura fundamentais como computação, armazenagem ou serviços de rede. O consumidor tem bastante controle das configurações desses componentes, inclusive podendo escolher e acessar o sistema operacional. Exemplos de ofertas IaaS são: AWS, OpenStack, DigitalOcean, Azure e Google Compute.

- **PaaS** (*Platform as a Service* ou Plataforma como Serviço): disponibiliza uma plataforma gerenciada onde o consumidor pode fazer *deploy* da sua aplicação. O consumidor geralmente não consegue controlar a infraestrutura diretamente, apenas as suas aplicações e possivelmente configurações do ambiente de execução. Exemplos de ofertas PaaS são: Heroku, Google App Engine, Elastic Beanstalk, OpenShift e Engine Yard.

- **SaaS** (*Software as a Service* ou Software como Serviço): disponibiliza um produto ou serviço cuja infraestrutura está rodando na nuvem. O consumidor não controla ou gerencia a infraestrutura nem mesmo a aplicação em si, apenas as suas configurações dentro do produto. Diversos serviços *online* podem ser considerados SaaS: GMail, Dropbox, Evernote, Basecamp, SnapCI e o TravisCI.

Existem diversos provedores públicos e privados de computação em nuvem. A Amazon foi a pioneira na área com o Amazon Web Services ou AWS (http://aws.amazon.com/) , oferecendo infraestrutura como serviço (IaaS) e uma grande oferta de produtos e soluções. Apesar de ser um serviço pago, ela oferece uma faixa de uso grátis no primeiro ano após você criar uma nova conta. Devido à sua flexibilidade e maturidade, vamos explorar como subir nosso ambiente de produção no AWS.

Configurando o ambiente no AWS

Criar uma nova conta no AWS é relativamente simples e exige que você tenha em mãos um celular e um cartão de crédito. Como eles lançam novos produtos e ofertas muito frequentemente, esse processo inicial de criação de conta está sempre mudando e ficando mais simples. Por esse motivo não vamos descrevê-lo passo a passo, mas vou supor que você já tenha uma conta configurada e ativada.

O principal serviço do AWS é o EC2, ou *Elastic Compute Cloud*, que nos provê unidades de computação na nuvem. Com ele você decide quanta capacidade de computação você precisa e provisiona os servidores desejados com uma chamada de API ou o clique de um botão. É uma forma rápida de conseguir um novo servidor na nuvem.

Com a conta AWS criada, você pode acessar o console de gerenciamento na URL https://console.aws.amazon.com/ e verá a tela principal mostrada na figura 8.1. Nessa tela você pode ter uma ideia da quantidade de serviços oferecidos pela Amazon. O EC2 é um dos links na seção *Compute & Networking*.
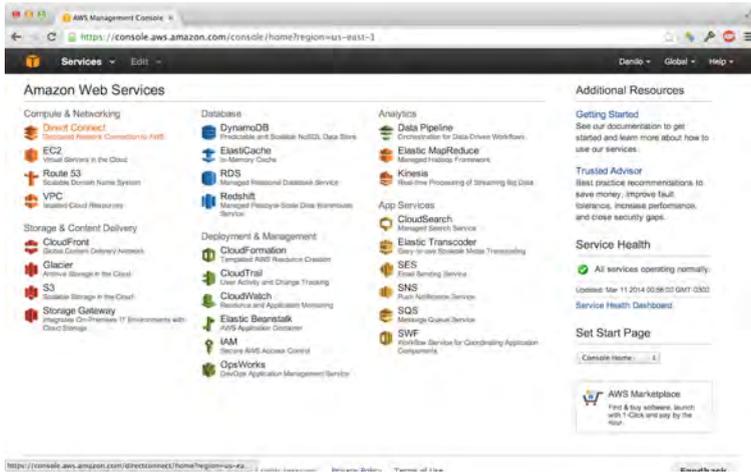
Figura 8.1: Console de controle AWS

Clicando no link do EC2 você é levado para o painel de controle desse serviço. A primeira coisa que precisamos fazer é criar um par de chaves SSH para conseguir acessar nossos servidores uma vez que forem criados. Para isso, acesse o link *"Key Pairs"* no menu lateral e clique no botão *Create New Pair* conforme mostrado na figura 8.2.
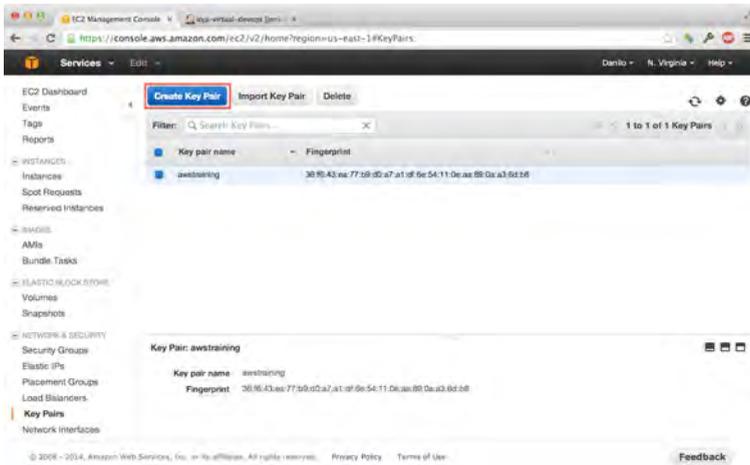
Figura 8.2: Criando um novo par de chaves de acesso SSH

Preencha o campo *Key pair name* com o nome da chave *devops* e clique em *Yes*. O arquivo da chave privada `devops.pem` será baixado conforme mostra a figura 8.3. Você precisa salvá-la em um lugar seguro pois a Amazon não guarda uma cópia desse arquivo para você, pois seria uma violação de segurança. Anote o caminho onde você salvar a chave primária, pois iremos usá-lo na próxima seção. Por enquanto, vamos presumir que o arquivo tenha sido salvo no seu diretório `HOME` em `~/devops.pem`. Com isso vamos conseguir acessar nossos servidores por SSH.
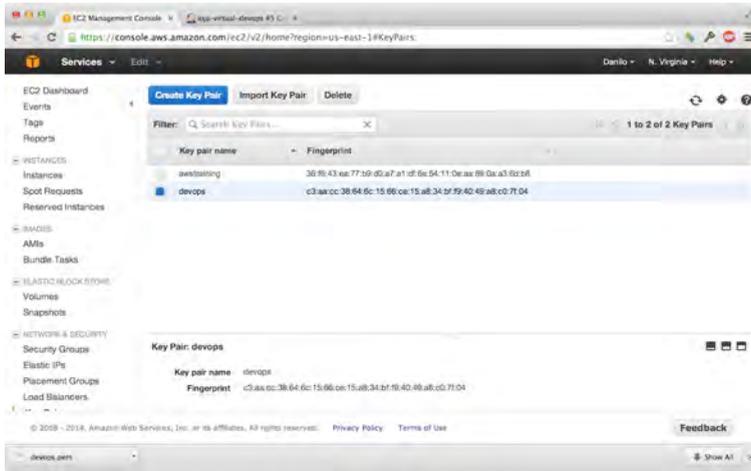
Figura 8.3: Par de chaves criado e download da chave primária

O próximo serviço que vamos utilizar é o VPC, ou *Virtual Private Cloud*. O VPC nos permite criar uma seção isolada do AWS onde podemos provisionar recursos e ter um pouco mais controle sobre a configuração e topologia da rede. Ao subir um servidor EC2 dentro de uma VPC, podemos escolher seu IP interno. Com esse serviço vamos conseguir manter a mesma topologia de rede que usamos para os nossos servidores locais no VirtualBox.

Para criar uma nuvem privada no VPC, clique no link *VPC* dentro do menu *Services* do console. No painel de controle do VPC, clique em *Get started creating a VPC* conforme mostra a figura 8.4.
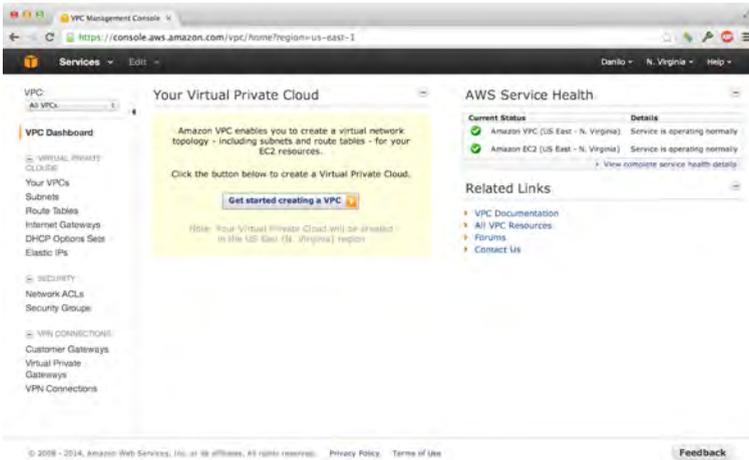
Figura 8.4: Painel de controle do VPC

Escolha a opção *VPC with a Single Public Subnet Only* conforme mostra a figura 8.5. Isso vai criar uma topologia de rede com um único segmento de rede público, com acesso direto à internet.

Figura 8.5: Criando uma nova VPC

Ao clicar em *Continue* você será levado para a próxima tela de configuração onde iremos definir nosso segmento de rede. Primeiramente vamos clicar no link *Edit VPC IP CIDR Block* e preencher o campo *IP CIDR block* com o valor `192.168.0.0/16`. Iremos colocar o mesmo valor no campo *Public Subnet* após clicar no link *Edit Public Subnet*, conforme mostra a figura 8.6.
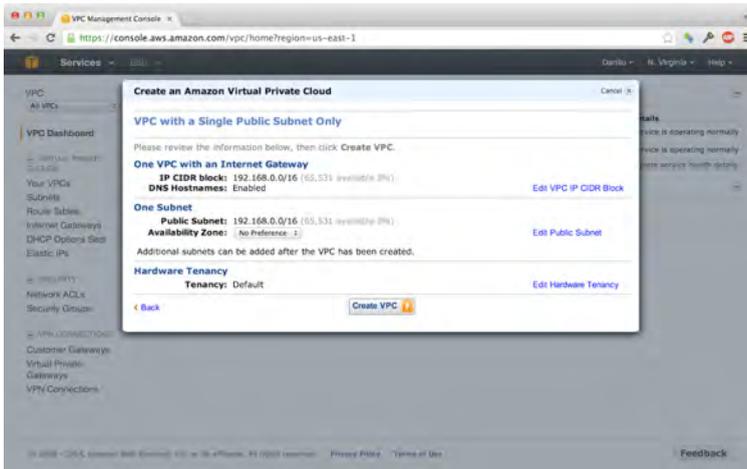
Figura 8.6: Configurações de segmentos de rede no VPC

Por fim, basta clicar em *Create VPC* para criar nossa nuvem privada e veremos uma mensagem de sucesso conforme mostra a figura 8.7.
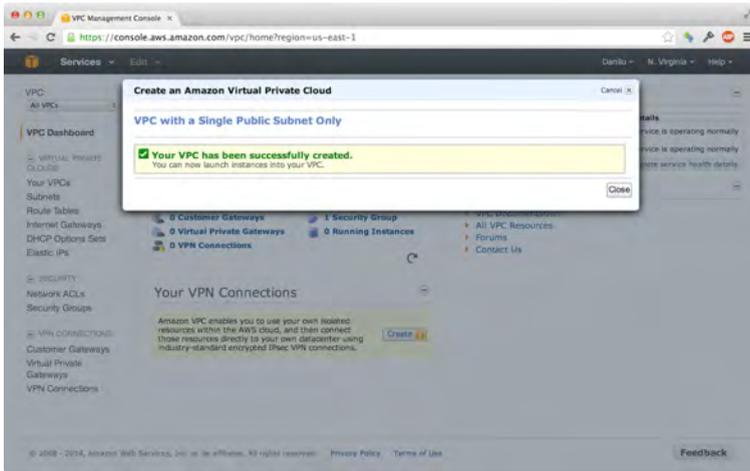
Figura 8.7: Nuvem privada no VPC criada com sucesso

Precisamos agora anotar o ID do nosso segmento de rede para uso na próxima seção. Para isto clique no link *Subnets* no menu lateral e escolha a *subnet* criada. Nesta tela você verá os detalhes do segmento de rede conforme mostra a figura 8.8. Anote o ID na sua tela que será diferente do que está destacado na figura uma vez que o AWS aloca um novo ID único sempre que um novo recurso for criado.
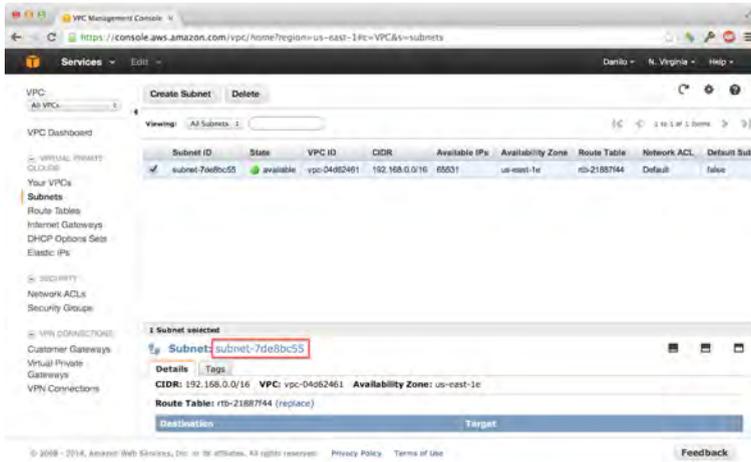
Figura 8.8: Anotando o ID da subnet criada

A última configuração que precisamos fazer no console do AWS é configurar um *security group*, ou grupo de segurança. Um grupo de segurança define as regras de acesso a instâncias EC2. Ele é uma espécie de *firewall* que bloqueia ou permite acesso para dentro ou fora da instância, dependendo de suas configurações.

Vamos clicar no link *Security Groups* no menu lateral e selecionar o grupo de segurança *default*. Nos detalhes, clique na aba *Inbound* para adicionar 4 novas regras para permitir acesso SSH, HTTP e HTTPS nas portas 22, 80/8080 e 443 respectivamente, conforme mostra a figura 8.9. Clique em *Apply Rule Changes* para que as mudanças tenham efeito.

Figura 8.9: Criando uma nova VPC

Provisionando os servidores de produção no AWS

Para provisionar os servidores de produção no AWS vamos usar um plugin do Vagrant chamado **vagrant-aws** (https://github.com/mitchellh/vagrant-aws) . As versões mais novas do Vagrant funcionam com provedores além do VirtualBox e esse plugin configura um provedor que sabe provisionar no AWS. Instalar o plugin é simples:

```
$ vagrant plugin install vagrant-aws
Installing the 'vagrant-aws' plugin. This can take a few
minutes...
Installed the plugin 'vagrant-aws (0.4.1)'!
```

Além disso, precisamos também importar uma *box* diferente que será usada pelo plugin. Conforme a documentação do plugin recomenda, chamaremos essa *box* de `dummy` e para adicioná-la basta executar o comando:

```
$ vagrant box add dummy \
> https://github.com/mitchellh/vagrant-aws/raw/master/dummy.box
Downloading box from URL: https://github.com/mitchellh/...
Extracting box...e: 0/s, Estimated time remaining: --:--:--)
Successfully added box 'dummy' with provider 'aws'!
```

Com o plugin instalado e o novo *box* disponível, podemos alterar nosso arquivo de configuração do Vagrant para adicionar as configurações de acesso ao AWS. Faremos isso alterando o arquivo `Vagrantfile`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"
  ...
  config.vm.provider :aws do |aws, override|
    aws.access_key_id = "<ACCESS KEY ID>"
    aws.secret_access_key = "<SECRET ACCESS KEY>"
    aws.keypair_name = "devops"

    aws.ami = "ami-a18c8fc8"
    aws.user_data = File.read("bootstrap.sh")
    aws.subnet_id = "<SUBNET ID>"
    aws.elastic_ip = true

    override.vm.box = "dummy"
    override.ssh.username = "ubuntu"
    override.ssh.private_key_path = "<PATH DA CHAVE PRIVADA>"
  end

  config.vm.define :db do |db_config|
    ...
    db_config.vm.provider :aws do |aws|
      aws.private_ip_address = "192.168.33.10"
      aws.tags = { 'Name' => 'DB' }
    end
  end

  config.vm.define :web do |web_config|
    ...
    web_config.vm.provider :aws do |aws|
      aws.private_ip_address = "192.168.33.12"
      aws.tags = { 'Name' => 'Web' }
    end
  end
```

```
  config.vm.define :monitor do |monitor_config| ... end

  config.vm.define :ci do |build_config|
    ...
    build_config.vm.provider :aws do |aws|
      aws.private_ip_address = "192.168.33.16"
      aws.tags = { 'Name' => 'CI' }
    end
  end
end
```

Você vai precisar fazer algumas substituições usando as suas próprias configurações do AWS que obtivemos na seção anterior. Os valores para `subnet_id` e `private_key_path` nós já temos anotados da seção anterior. Para conseguir os valores do `access_key_id` e do `secret_access_key`, você pode acessar o link *Security Credentials* no menu de usuário mostrado na figura 8.10.

Figura 8.10: Menu para acessar credenciais de segurança

Na próxima tela, ao expandir a seção *Access Keys* você verá um botão *Create New Access Key* conforme mostra a figura 8.11.
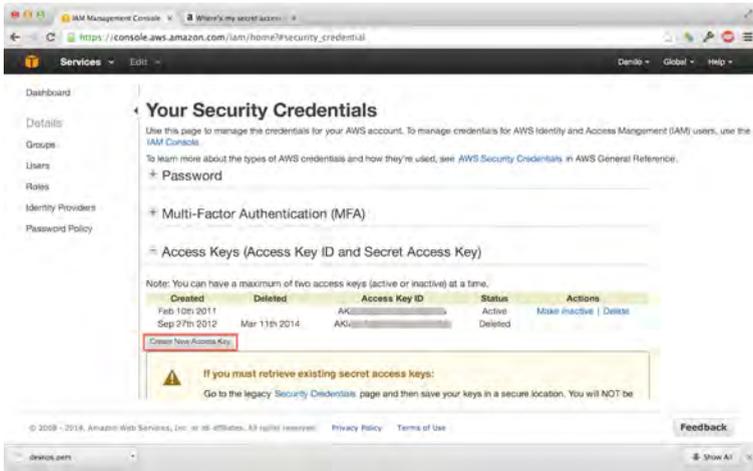
Figura 8.11: Menu para acessar credenciais de segurança

Após criar o *access key* com sucesso você verá uma nova tela que lhe dará os valores para preencher nos campos `access_key_id` e `secret_access_key` conforme mostra a figura 8.12. Essas são suas credenciais pessoais, por isso não as compartilhe com ninguém. Se alguém obtiver suas credenciais, ele conseguirá subir máquinas e você terá que pagar a conta!

Figura 8.12: Menu para acessar credenciais de segurança

No `Vagrantfile` definimos algumas outras configurações do provedor AWS. O `keypair_name` é o nome do par de chaves que criamos na seção anterior. O `ami` é o ID da imagem base que o AWS usará para criar um novo servidor. Nesse caso, o ID `"ami-a18c8fc8"` corresponde a uma imagem base do Ubuntu 12.04 LTS de 32 bits, a mesma versão do sistema operacional e arquitetura usados nas máquinas virtuais do VirtualBox.

Por fim, o parâmetro `user_data` aponta para um novo arquivo que precisamos criar. Ao subir uma nova instância no EC2, você pode passar um pequeno *script* que será executado assim que a máquina subir pela primeira vez. Nesse caso, precisamos de um *script* para instalar o Puppet em si, que não vem pré-instalado na AMI que escolhemos. O conteúdo desse novo arquivo `bootstrap.sh` deve ser:

```
#!/bin/sh
set -e -x
export DEBIAN_FRONTEND=noninteractive

wget https://apt.puppetlabs.com/puppetlabs-release-precise.deb
dpkg -i puppetlabs-release-precise.deb
```

```
apt-get -y update
apt-get -y install puppet-common=2.7.19* facter=1.7.5*
```

Esse *script* é relativamente simples. Os comandos `wget` e `dpkg -i` configuram um novo repositório de pacotes com todos os *releases* do Puppet mantidos pela própria Puppet Labs. Com isso, simplesmente usamos o comando `apt-get` para atualizar seu índice de busca e instalar os pacotes do Puppet na versão 2.7.19 e do Facter na versão 1.7.5, as mesmas que estamos usando nas máquinas virtuais locais do VirtualBox.

Pronto, agora podemos provisionar nossos servidores de produção no AWS com o Vagrant usando uma nova opção `--provider=aws`, começando pelo servidor `db`:

```
$ vagrant up db --provider=aws
Bringing machine 'db' up with 'aws' provider...
==> db: Installing Puppet modules with Librarian-Puppet...
==> db: Warning! The AWS provider doesn't support any of the
==> db: Vagrant high-level network configurations
==> db: (`config.vm.network`). They will be silently ignored.
==> db: Launching an instance with the following settings...
==> db:  -- Type: m1.small
==> db:  -- AMI: ami-a18c8fc8
...
notice: Finished catalog run in 39.74 seconds
```

Caso veja uma mensagem de erro dizendo que o servidor já está provisionado com um provedor diferente, simplesmente faça um `vagrant destroy db ci web` para destruir as suas máquinas virtuais locais. Em futuras versões do Vagrant você vai poder subir o mesmo servidor usando diferentes provedores ao mesmo tempo. Após destruí-las, você pode executar o comando `vagrant up db --provider=aws` novamente e ele irá funcionar conforme mostrado anteriormente.

Temos nosso primeiro servidor rodando na nuvem! O próximo será o servidor `ci`, pois precisamos publicar o pacote `.deb` que será usado pelo servidor `web` posteriormente. Mas antes de fazer isso, vamos fazer uma pequena alteração na classe `loja_virtual::repo` pois no EC2 teremos apenas uma interface de rede `eth0` e nosso módulo assume que o repositório

será acessível no endereço IP da interface `eth1`. Fazemos esta alteração no final do arquivo `modules/loja_virtual/manifest/repo.pp`:

```
class loja_virtual::repo($basedir, $name) {
  package { 'reprepro': ... }
  $repo_structure = [...]
  file { $repo_structure: ... }
  file { "$basedir/conf/distributions": ... }
  class { 'apache': }

  if $ipaddress_eth1 {
    $servername = $ipaddress_eth1
  } else {
    $servername = $ipaddress_eth0
  }

  apache::vhost { $name:
    port       => 80,
    docroot    => $basedir,
    servername => $servername,
  }
}
```

Aqui estamos usando uma nova sintaxe do Puppet para expressões condicionais. Se a variável `$ipaddress_eth1` estiver definida, ela será usada, caso contrário usaremos `$ipaddress_eth0`. Isso permite que nosso código Puppet continue funcionando localmente com as máquinas virtuais do VirtualBox. Podemos agora subir o servidor `ci` na nuvem:

```
$ vagrant up ci --provider=aws
Bringing machine 'ci' up with 'aws' provider...
==> ci: Installing Puppet modules with Librarian-Puppet...
==> ci: Warning! The AWS provider doesn't support any of the
==> ci: Vagrant high-level network configurations
==> ci: (`config.vm.network`). They will be silently ignored.
==> ci: Launching an instance with the following settings...
==> ci:  -- Type: m1.small
==> ci:  -- AMI: ami-a18c8fc8
...
notice: Finished catalog run in 423.40 seconds
```

Com o servidor provisionado, o Jenkins irá iniciar um novo *build* da loja virtual e tentará publicar um novo pacote `.deb`. Se lembrarmos o que foi preciso fazer no capítulo 7 para configurar o repositório, vamos precisar logar na máquina para gerar o par de chaves GPG e exportar a chave pública. Caso não lembre de todos os passos, releia o capítulo anterior, se lembrar, siga as mesmas instruções mostradas adiante:

```
$ vagrant ssh ci
ubuntu@ip-192-168-33-16$ sudo apt-get install haveged
Reading package lists... Done
...
Setting up haveged (1.1-2) ...
ubuntu@ip-192-168-33-16$ sudo su - jenkins
jenkins@ip-192-168-33-16$ gpg --gen-key
gpg (GnuPG) 1.4.11; Copyright (C) 2010 ...
...
pub   2048R/3E969140 2014-03-11
      Key fingerprint =
        2C95 5FF8 9789 14F9 D6B3  ECD4 4725 6390 ...
uid                  Loja Virtual <admin@devopsnapratica.com.br>
sub   2048R/67432481 2014-03-11
jenkins@ip-192-168-33-16$ gpg --export --armor \
> admin@devopsnapratica.com.br > /var/lib/apt/repo/
                                               devopspkgs.gpg
jenkins@ip-192-168-33-16$ logout
ubuntu@ip-192-168-33-16$ logout
Connection to 54.84.207.166 closed.
$
```

Não se esqueça de anotar o ID da chave pública gerada, nesse casso `3E969140`, e substituir o parâmetro `key` do recurso `Apt::Source[devopsnapratica]` na classe `loja_virtual::web`. Agora basta esperar que o nosso *build* termine e publique o pacote `.deb`. Para acompanhar o progresso do *build*, você precisa descobrir o nome do host público do servidor Jenkins no AWS. Para isto, acesse o link *Instances* no menu lateral do painel de controle do EC2, selecione o servidor "CI" e copie o valor do campo *Public DNS* conforme mostra a figura 8.13.
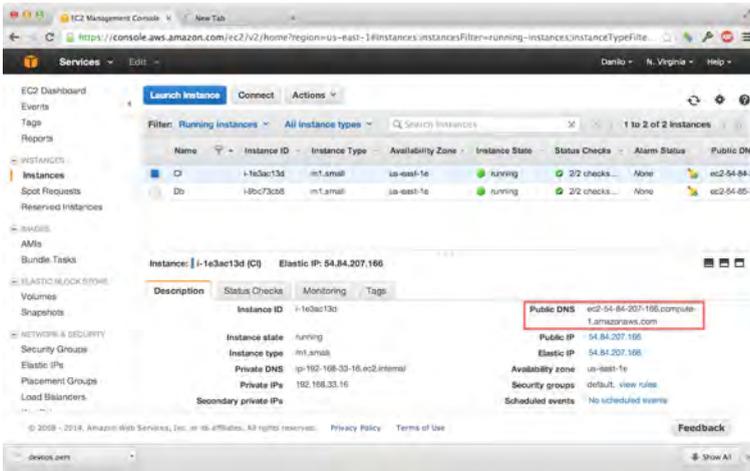
Figura 8.13: Nome do host público do servidor de CI

Com o endereço em mãos, acesse o Jenkins em uma nova janela do navegador, usando a porta 8080. Nesse caso, nossa URL é http://ec2-54-84-207-166.compute-1.amazonaws.com:8080/ e você verá o Jenkins rodando na nuvem conforme mostra a figura 8.14.
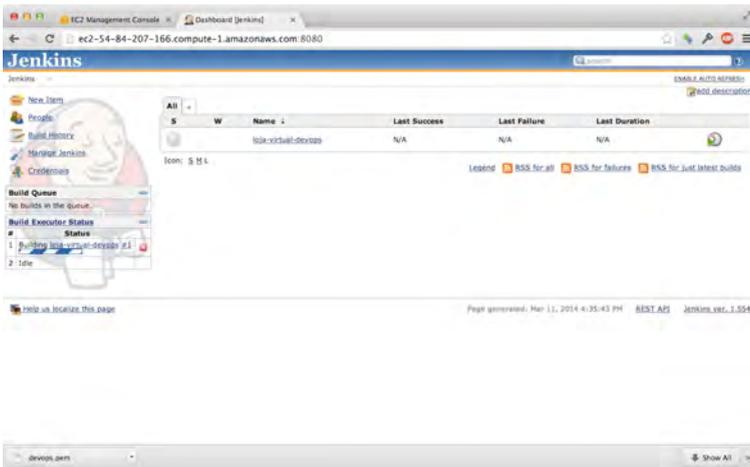
Figura 8.14: Jenkins rodando na nuvem

Assim que o *build* completar com sucesso, você pode provisionar o servidor `web`:

```
$ vagrant up web --provider=aws
Bringing machine 'web' up with 'aws' provider...
==> web: Installing Puppet modules with Librarian-Puppet...
==> web: Warning! The AWS provider doesn't support any of the
==> web: Vagrant high-level network configurations
==> web: (`config.vm.network`). They will be silently ignored.
==> web: Launching an instance with the following settings...
==> web:  -- Type: m1.small
==> web:  -- AMI: ami-a18c8fc8
...
notice: Finished catalog run in 88.32 seconds
```

Você pode usar o mesmo procedimento para descobrir o nome do host público do servidor `web` no AWS, conforme mostra a figura 8.15.
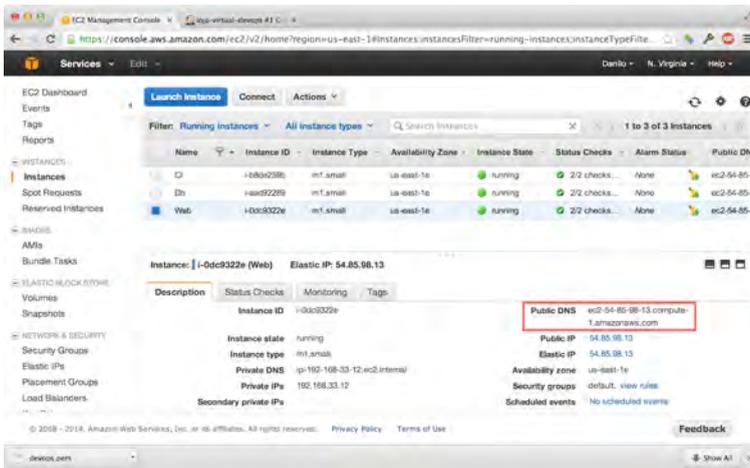
Figura 8.15: Nome do host público do servidor web

No nosso caso, ao acessar http://ec2-54-85-98-13.compute-1.amazonaws.com:8080/devopsnapratica/ veremos nossa loja virtual rodando na nuvem, conforme mostra a figura 8.16. Sucesso! Conseguimos migrar nossa aplicação

para a nuvem sem muita dor de cabeça. O investimento em automação do processo valeu a pena.
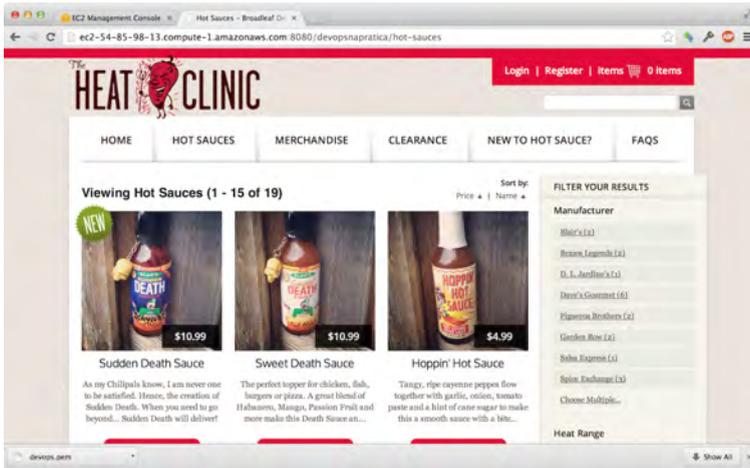
Figura 8.16: Loja virtual rodando na nuvem

Como os recursos no AWS são pagos, precisamos destruir os servidores para economizar dinheiro. **Cuidado! Se você esquecer este passo, você receberá uma conta inesperada no final do mês!**

```
$ vagrant destroy ci web db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Terminating the instance...
==> db: Running cleanup tasks for 'puppet' provisioner...
    web: Are you sure you want to destroy the 'web' VM? [y/N] y
==> web: Terminating the instance...
==> web: Running cleanup tasks for 'puppet' provisioner...
    ci: Are you sure you want to destroy the 'ci' VM? [y/N] y
==> ci: Terminating the instance...
==> ci: Running cleanup tasks for 'puppet' provisioner...
```

Para garantir que as instâncias foram realmente terminadas, acesse o link *"Instances"* no menu lateral do painel de controle do EC2 e você deve ver algo parecido com a figura 8.17.
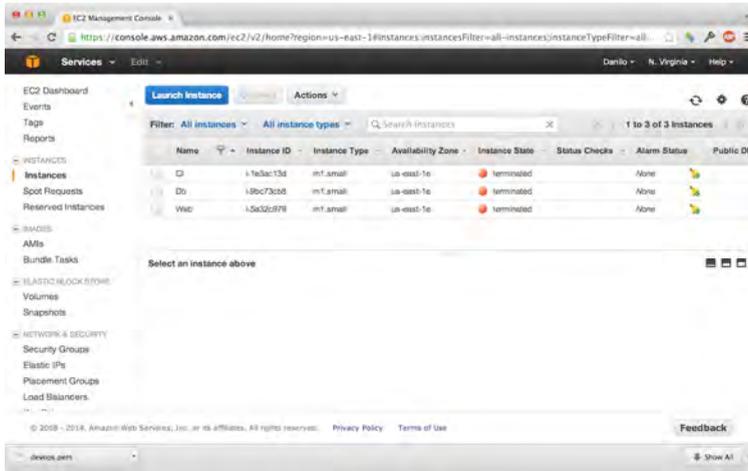
Figura 8.17: Instâncias EC2 terminadas

8.2 DevOps além das ferramentas

Seria impossível cobrir todas as ferramentas e técnicas que estão evoluindo no mundo DevOps neste livro. Para todos os tópicos abordados, existem diversas ferramentas alternativas sendo usadas pela comunidade. Além disso, existem os tópicos que não foram abordados e suas respectivas opções de ferramentas.

O motivo de termos usado as ferramentas que usamos foi para mostrar um exemplo prático de entrega contínua e DevOps. DevOps é mais do que suas escolhas de ferramentas. John Willis e Damon Edwards definiram o acrônimo **CAMS** para definir o que é DevOps [19] e Jez Humble mais tarde introduziu *Lean* para completar o acrônimo **CALMS**:

- **(C)ultura**: pessoas e processos são mais importantes. Se a cultura não estiver presente, qualquer tentativa de automação está destinada a falhar.

- **(A)utomação**: libere os humanos para realizar tarefas que exigem criatividade e intuição e deixe as tarefas repetitivas para os computadores, que sabem executá-las rapidamente e de forma bem mais confiável.

- **(L)ean (pensamento enxuto)**: diversos princípios *lean* influenciam a cultura DevOps, como a melhoria contínua, o foco em qualidade, a eliminação de desperdícios, a otimização do todo e o respeito às pessoas.
- **(M)edição**: se você não souber medir, não saberá avaliar se está melhorando ou piorando. Uma adoção de DevOps de sucesso irá medir tudo o que for possível, por exemplo: métricas de desempenho, métricas de processo, métricas de negócio etc.
- **(S)haring (compartilhamento)**: a colaboração e o compartilhamento de ideias e conhecimento ajudam a criar a cultura necessária para o sucesso com DevOps.

Com essas características em mente, vamos discutir alguns dos tópicos que não foram abordados no livro para que você prossiga com seus estudos sobre DevOps. Não se esqueça do último princípio: compartilhe o que você aprender e contribua com a comunidade! Compartilhar histórias é uma das melhores maneiras de criar conhecimento.

8.3 Sistemas avançados de monitoramento

O sistema de monitoramento que configuramos no capítulo 3 foca na detecção e notificação de falhas no ambiente de produção. O Nagios é uma ferramenta bastante comum na indústria, porém tem suas limitações: a interface gráfica é antiquada e difícil de usar, suas arquivos de configuração são complicados e não-intuitivos e em um ambiente mais dinâmico onde servidores podem ir e vir a qualquer momento, o Nagios não é uma ferramenta muito flexível.

Existem outros aspectos importantes que devem ser considerados em um sistema de monitoramento mais completo. Em particular, algumas áreas estão recebendo bastante atenção da comunidade e diversas ferramentas e produtos estão surgindo para atacá-las:

- **Agregação de *logs***: quanto mais servidores você possuir, mais difícil é investigar e depurar problemas. A abordagem tradicional de logar no servidor e fazer um `grep` nos arquivos não funciona pois você precisa ficar conectando em diversos servidores para depurar um único

problema. Por esse motivo, diversas ferramentas de agregação e indexação de logs estão surgindo. Um conjunto de ferramentas de código livre tem ganhado popularidade nessa área: o **Logstash** (http://logstash.net/) coleta, filtra, analisa e armazena entradas de *log* de diversas fontes diferentes no **Elasticsearch** (http://elasticsearch.org/) , uma ferramenta de busca. Outras opções nesse espaço são o **Graylog2** (http://graylog2.org/) , o **Splunk** (http://www.splunk.com/) , o **Loggly** (https://www.loggly.com/) , o **PaperTrail** (https://papertrailapp.com/) e o **Logentries** (https://logentries.com/) .

- **Métricas**: eu me lembro que alguém me disse uma vez que "você não tem um problema de desempenho, você tem um problema de visibilidade". Se você enxergar onde e quando o problema acontece vai ser muito mais fácil resolvê-lo. Por isso DevOps encoraja você a medir o máximo possível. Métricas podem ser de sistema, da aplicação, do processo, de negócio ou até das pessoas. Existem algumas ferramentas evoluindo neste espaço que permitem a captura e armazenamento de métricas. A biblioteca **Metrics** (http://metrics.codahale.com/) escrita em Java pelo Coda Hale e com algumas variações em outras linguagens provê uma boa API para que desenvolvedores passem a publicar diversos tipos de métrica a partir do código. No lado da agregação e armazenamento existem ferramentas como o **StatsD** (https://github.com/etsy/statsd/) , o **CollectD** (http://collectd.org/) , o **Carbon** (http://graphite.wikidot.com/carbon) , o **Riemann** (http://riemann.io/) , o **Ganglia** (http://ganglia.sourceforge.net/) , o **TempoDB** (https://tempo-db.com/) , o **OpenTSDB** (http://opentsdb.net/) e o **Librato** (https://metrics.librato.com/) .

- **Visualizações**: quanto mais dados e métricas você coletar, maior é o risco de você sofrer uma sobrecarga de informações. Por isso é importante investir em ferramentas e técnicas de visualização de informação para sintetizar grandes volumes de dados em um formato mais digestível para humanos. Algumas ferramentas nesse espaço são o **Graphite** (http://graphite.wikidot.com/) , o **Cubism.js** (http://square.github.io/cubism/) , o **Tasseo** (https://github.com/obfuscurity/tasseo)

e o **Graphene** (http://jondot.github.io/graphene/) .

- **Informações em tempo de execução**: para entender onde estão os gargalos da sua aplicação em produção, você precisa coletar informações em tempo de execução. Com isso você pode descobrir quanto tempo seu código está gastando em chamadas externas, renderizando páginas HTML ou quais consultas estão levando muito tempo no banco de dados. Outra área que tem ganhado importância é a captura de dados do ponto de vista de usuário, por exemplo: quanto tempo a sua página demora para carregar no navegador do usuário. Ferramentas nesse espaço são o **NewRelic** (http://newrelic.com/) , o **Google Analytics** (http://www.google.com/analytics/) e o **Piwik** (http://piwik.org/) . Além disso, é importante também capturar erros com ferramenta como o **NewRelic** ou o **Airbrake** (https://airbrake.io/) .

- **Monitoramento de disponibilidade**: outro tipo de monitoramento é saber quando a sua aplicação está no ar ou não. Se você tiver usuários distribuídos ao redor do mundo, esse tipo de monitoramento é ainda mais importante, pois você pode estar indisponível em apenas alguns lugares e não outros. Ferramentas nesse espaço são o **Pingdom** (https://www.pingdom.com/) , o **NewRelic**, o **Monitor.us** (http://www.monitor.us/) e o **Uptime Robot** (http://uptimerobot.com/) .

- **Sistemas analíticos**: todos esses dados podem ser consumidos por um sistema analítico para tentar encontrar correlações, tendências, ou *insights* avançados sobre seu sistema. Ainda não existem muitas ferramentas nesse espaço, porém o **NewVem** (http://www.newvem.com/) possui alguns relatórios analíticos interessantes sobre uso de recursos na nuvem para otimizar seus custos.

Soluções de monitoramento mais holísticas tentam atacar todas essas áreas, permitindo que você monitore em diversos níveis e agregue informações em um único lugar. Com isso você pode detectar correlações entre dados independentes. O **NewRelic** (http://newrelic.com/) é um produto SaaS pago que está tentando ir nessa direção.

Quanto mais avançado e completo for o seu sistema de monitoramento, maior vai ser sua confiança para fazer mais *deploys* em produção. A detecção de problemas operacionais se torna uma extensão – e em alguns casos até uma substituição – do seu processo de testes. A IMVU, por exemplo, realiza mais de 50 *deploys* por dia e criou um sistema de imunidade que monitora diversas métricas durante um *deploy*. Quando algum problema de regressão é detectado, esse sistema reverte o último *commit* automaticamente e envia alertas para proteger o sistema em produção [5].

8.4 Pipelines de entrega complexas

No nosso exemplo do capítulo 7 configuramos uma pipeline de entrega bem simplificada. No mundo real, é comum que existam mais estágios e ambientes pelos quais uma aplicação precisa passar antes de chegar em produção. Em casos mais complexos, você terá diversas aplicações, componentes e serviços que se integram entre si. Nessas situações, você vai precisar de mais níveis de teste de integração para garantir que os componentes e serviços se integram corretamente.

Criar e gerenciar pipelines mais complexas com o Jenkins começa a ficar mais complicado conforme o número de *jobs* cresce. A pipeline não é um conceito primário do Jenkins, mas pode ser modelada usando dependências e gatilhos entre *jobs*, como fizemos no capítulo anterior. Nesse espaço, a ferramenta **Go** (http://www.go.cd/) escrita pela ThoughtWorks teve seu código recentemente aberto e foi escrita com a pipeline de entrega em primeiro plano. Com um modelo de execução de *jobs* distribuído e uma forma de modelar diferentes ambientes, o Go é uma opção atraente para gerenciar pipelines complexas.

8.5 Gerenciando mudanças no banco de dados

Um dos assuntos que não abordamos mas é muito importante é como gerenciar **mudanças no banco de dados**. Na nossa aplicação, o Hibernate cria o *schema* inicial de tabelas e preenche com os dados de referência da loja virtual. No mundo real, o *schema* do banco de dados irá evoluir e você não pode simplesmente apagar e reconstruir os dados a cada *deploy*.

Para lidar com a evolução do banco de dados, geralmente tentamos desacoplar o processo de *deploy* da aplicação do processo de *deploy* do banco de dados. Por exigir mais cuidado, algumas empresas optam por atualizar o banco de dados uma vez por semana enquanto a aplicação vai para produção diversas vezes por dia. Práticas para evoluir o *schema* de um banco de dados de forma segura estão descritas nos livros de Pramos Sadalage "Refatorando banco de dados" [9] e "Receitas para integração contínua de banco de dados" [15]. O livro "Entrega contínua" [6] também tem um capítulo dedicado ao gerenciamento de dados.

8.6 Orquestração de deploy

Nosso processo de *deploy* é bem simples. Ele simplesmente reprovisiona o servidor `web` automaticamente sempre que um novo *commit* passar pela pipeline. Quanto mais componentes você possuir na sua arquitetura, maior vai ser a necessidade de orquestrar a ordem do *deploy*. No nosso caso, por exemplo, precisaríamos provisionar o banco de dados antes do servidor web. Se você tiver um *cluster* replicado do banco de dados, você vai querer fazer o *deploy* em uma determinada ordem para evitar a perda de dados e diminuir o tempo em que o banco fica indisponível.

Para essas situações, é recomendado criar um *script* (ou um conjunto de *scripts*) responsável por decidir a ordem em que o *deploy* deve acontecer. Ferramentas como o **Capistrano** (http://capistranorb.com/) , o **Fabric** (http://fabfile.org/) ou até mesmo *scripts* shell podem ser usadas para esse tipo de orquestração. Outra alternativa no mundo Puppet é usar o **MCollective** (http://puppetlabs.com/mcollective) , um produto pago oferecido pela Puppet Labs.

Uma técnica importante para reduzir o risco dos seus *deploys* é conhecida como ***blue-green deployments*** – ou implantações azul-verde. Se chamarmos o ambiente de produção atual de "azul", a técnica consiste em introduzir um ambiente paralelo "verde" com a nova versão do software e, uma vez que tudo é testado e está pronto para começar a operar, você simplesmente redireciona todo o tráfego de usuários do ambiente "azul" para o ambiente "verde". Em um ambiente de computação em nuvem, assim que for confirmado que

o ambiente ocioso não é mais necessário, é comum descartá-lo, uma prática conhecida como **servidores imutáveis**.

8.7 Gerenciando configurações por ambiente

É comum que sua pipeline de entrega passe por diversos ambientes, por exemplo: um ambiente de testes, um ambiente de homologação, um ambiente pré-produção e o ambiente de produção. Um princípio importante é que você deve **construir seus artefatos uma vez e usá-los em todo lugar**. Isso significa que você não deve colocar configurações específicas de um ambiente no artefato principal da aplicação.

Um exemplo desse tipo de configuração é a URL da conexão com o banco de dados. Em cada ambiente você terá uma instância diferente do banco de dados rodando, portanto você precisa configurar a aplicação para apontar para o banco de dados correto. Existem algumas abordagens diferentes para resolver este tipo de problema:

- **Sistema de controle de versões**: esta é uma abordagem com a qual você comita um arquivo de configuração diferente para cada ambiente no sistema de controle de versões. No mundo Java, é comum você criar arquivos `.properties` para cada ambiente. Sua aplicação precisa então saber carregar o arquivo certo quando estiver rodando. A desvantagem é que todas as configurações ficam dentro do artefato da aplicação, tornando difícil alterá-las sem fazer um novo *deploy*.

- **Pacotes de configuração por ambiente**: esta abordagem é parecida com a anterior, porém, em vez de empacotar os arquivos de configuração no artefato da aplicação, você gera pacotes de configuração diferentes com as configurações específicas para cada ambiente. Isso permite reconfigurar sem fazer um novo *deploy* da aplicação.

- **Configuração com DNS**: ao invés de mudar a URL em cada ambiente, você pode usar a mesma URL em todos os ambientes – por exemplo, `bd.app.com` – e usar o sistema de resolução de IPs do DNS para resolver o mesmo nome de *host* para o IP correto naquele ambiente. Essa

abordagem só funciona para configurações do tipo URL e não funcionaria para outros tipos de configurações.

- **Provedor externo de configurações**: em alguns casos, você pode querer controlar esse tipo de configuração em um lugar centralizado. Um conceito comum no mundo de operações é um CMDB (*Configuration Management DataBase* ou banco de dados para gerenciamento de configurações), um repositório de todos os ativos de TI da sua organização e suas respectivas configurações. Existem algumas formas diferentes de implementar essa prática: no Puppet, o **hiera** (https://github.com/puppetlabs/hiera) armazena configurações de forma hierárquica, permitindo sobrescrever valores por ambiente e distribuí-lo para cada *host* em conjunto com o Puppet master; no Chef você pode criar *data bags* para armazenar configurações ou implementar um LWRP (*LightWeight Resource Provider* ou provedor de recursos leves) para obter configurações de um servidor central externo. Outra forma de implementar essa prática é utilizar o próprio sistema de controle de versões e suas habilidades de *branching* e *merging*, conforme descrito por Paul Hammant [11]; algumas empresas acabam implementando uma solução customizada para resolver esse problema.

- **Variáveis de ambiente**: uma abordagem usada no Heroku que se tornou popular quando um de seus engenheiros, Adam Wiggins, publicou o artigo sobre a *"twelve-factor app"* [18]. Nessa abordagem toda configuração que varia entre um ambiente e outro é definida em variáveis de ambiente. Isso garante que nenhuma configuração acabe acidentalmente no repositório de código e é um padrão agnóstico da linguagem e do sistema operacional em que a aplicação vai rodar.

Existem vantagens e desvantagens de cada abordagem então você vai precisar considerar qual delas faz mais sentido no seu ambiente. Por exemplo, o Heroku exige que configurações sejam fornecidas em variáveis de ambiente, então você não tem outra opção. Caso você esteja controlando toda a sua infraestrutura com o Puppet, a abordagem de usar o hiera se torna mais interessante.

8.8 Evolução arquitetural

Ao final do livro, terminamos com um ambiente composto de quatro servidores: os servidores `web` e `db` são parte do ambiente de produção essenciais para que usuários consigam acessar a loja virtual. Os servidores `monitor` e `ci` são parte da nossa infraestrutura de entrega e monitoramento. Da mesma forma que aprendemos e melhoramos nosso *design* de código depois que o código está escrito e em uso, vamos aprender e ter feedback sobre nossa infraestrutura uma vez que a aplicação está em produção.

Conforme você investe no seu sistema de monitoramento e obtém métricas, dados e informações sobre como sua aplicação se comporta em produção, você terá *feedback* para melhorar a arquitetura da sua infraestrutura. Por exemplo, a funcionalidade de busca da nossa loja virtual é fornecida pelo Solr e no momento estamos rodando uma instância embutida do Solr, na mesma JVM. Conforme a quantidade de produtos indexados aumenta, o Solr irá consumir mais memória e irá competir por recursos com o próprio Tomcat. No pior caso, um problema no Solr pode afetar a aplicação toda.

Uma evolução da arquitetura seria migrar o Solr para um servidor dedicado. Nesse caso, quando algum problema acontecer, a aplicação continua rodando. Você apenas fornece uma degradação da funcionalidade de busca. Dependendo da importância dessa funcionalidade, você pode escolher simplesmente desabilitá-la ou usar uma implementação alternativa que usa o banco de dados. Todas essas são decisões válidas e tornam a sua aplicação mais resiliente do ponto de vista do usuário.

Quando estiver planejando sua arquitetura física e a capacidade esperada para sua aplicação, você deve levar em conta diversos fatores, geralmente associados com requisitos não-funcionais:

- **Frequência de mudanças**: mantenha junto o que muda na mesma frequência. Este princípio se aplica tanto no *design* de software quanto na arquitetura do seu sistema. Se algum componente tem mais de um motivo para mudar, é uma indicação de que ele deve ser dividido. No nosso caso, essa foi uma das motivações para separarmos os estágios de *build*, os pacotes e os repositórios de código da aplicação e da infraestrutura.

- **Pontos de falha**: é importante conhecer os pontos de falha da sua arquitetura. Se você não escolhê-los de forma propositada, eles ainda existirão. Você só não vai saber onde e quando uma falha pode acontecer. No exemplo do Solr embutido, sabemos que uma falha nesse componente pode derrubar a aplicação inteira, portanto algumas decisões arquiteturais devem ser definidas com cenários de falha em mente.

- **Unidades de *deploy* e dependências**: existe um princípio de arquitetura que diz que é melhor ter "pequenas peças, fracamente acopladas" do que ter um único monólito onde todas as peças estão altamente acopladas. Ao pensar na sua arquitetura, considere quais componentes devem ser isolados do resto e quais podem ir para produção de forma independente do resto. No nosso caso, esta foi outra motivação para separar o servidor `db` do servidor `web`.

- **Acordos de nível de serviço** (*service-level agreements* ou SLAs): um SLA é uma métrica que define como um serviço deve operar dentro de limites predefinidos. Em alguns casos, provedores colocam penalizações financeiras para dar mais peso ao SLA. SLAs geralmente cobrem características como desempenho e disponibilidade. Quando a sua aplicação provê um SLA, você vai precisar tomar decisões arquiteturais para tentar garantir que você consegue atendê-lo. Por exemplo, se você fornece uma API pública que precisa estar disponível 99.99% por ano – isso significa que ela só pode estar indisponível no máximo 52 minutos e 33 segundos por ano – você pode precisar adicionar camadas para limitar o número de requisições que um cliente pode fazer, para que nenhum cliente consiga monopolizar seus recursos ou derrubem sua API. John Allspaw cobre esse assunto em bem mais detalhes no livro "The Art of Capacity Planning" [1].

- **Escalabilidade**: dependendo das suas expectativas sobre escalabilidade, você vai precisar decidir os pontos de extensão na sua arquitetura. Existem duas abordagens comuns para lidar com isso: **escalabilidade horizontal** permite que você adicione capacidade colocando mais servidores além dos que você já possui; **escalabilidade vertical** permite que você adicione capacidade colocando mais recursos – como CPU,

memória, disco, rede etc. – nos servidores que você já possui. Quando você tem componentes separados, você ganha a possibilidade de considerar esses requisitos de escalabilidade em cada componente. Por exemplo, você pode escolher escalar seus servidores web horizontalmente, enquanto seu banco de dados irá escalar verticalmente.

- **Custos**: o outro lado da moeda ao separar mais e mais componentes é a questão do custo. Em ambientes de computação em nuvem, você pode otimizar seus custos conforme você aprende mais sobre como sua aplicação se comporta em produção. No entanto, se você está planejando comprar seu próprio hardware, não esqueça de considerar os custos de cada opção arquitetural antes de tomar qualquer decisão.
- **Padrões de uso**: se você sabe que sua aplicação tem picos de uso previsíveis, ou características sazonais bem conhecidas, é possível otimizar sua infraestrutura para reduzir custos. Isto é importante princalmente em um ambiente de computação em nuvem, onde é fácil aumentar ou diminuir o número de servidores. No nosso caso, o servidor de integração contínua provavelmente não precisa estar disponível 24 horas por dia, 7 dias por semana. Se nossos desenvolvedores trabalham no mesmo escritório, é bem capaz que o servidor não seja utilizado fora do horário comercial ou nos fins de semana. Esta é uma vantagem de migrar sua infraestrutura para a nuvem, pois você pode simplesmente desligar o servidor e parar de pagar durante o período ocioso.
- **Uso de recursos**: por fim, outra característica que deve ser considerada quando você decidir a sua arquitetura é o padrão de uso de recursos de cada componente. Servidores web geralmente precisam usar mais CPU e concorrência para atender diversas requisições simultâneas. Por outro lado, servidores de banco de dados geralmente usam mais memória e disco para indexação e persistência. Conhecer essas características de cada componente do seu sistema é importante para a escolha da melhor arquitetura.

Não existe uma resposta correta para todas as situações. Conforme você ganha mais experiência, você aprende a identificar quais características são

mais importantes para sua aplicação. O princípio da melhoria contínua se aplica aqui, pois é uma maneira de avaliar se as suas mudanças arquiteturais estão trazendo os benefícios desejados.

8.9 Segurança

Um assunto bastante importante em vários aspectos da sua aplicação e infraestrutura é a segurança. Durante o livro, nós escolhemos senhas fracas e às vezes até as deixamos em branco para simplificar o exemplo. No entanto, essa nunca é a melhor opção em um ambiente de produção. Outro aspecto importante é que você não deve compartilhar segredos – senhas, *passphrases*, certificados ou chaves primárias – entre os diferentes ambientes. Em alguns casos é importante inclusive que você não compartilhe esses segredos com todos os membros da equipe.

Nunca coloque segredos em texto puro no repositório de código. Como regra geral, sempre tente criptografar informações sensíveis para que, mesmo quando alguém tenha acesso à informação, ele não seja capaz de obter o segredo. Nessa mesma linha de raciocínio, é importante que você não armazene informações pessoais dos seus usuários em texto puro. Não deixe que segredos apareçam em texto puro nos seus *logs*. Não compartilhe senhas por e-mail ou anote em um *post-it* que fica do lado do servidor. Seu sistema é tão seguro quanto seu ponto mais fraco e, por incrível que pareça, o ponto fraco é geralmente humano.

Sempre que precisamos gerar algum tipo de chave, perceba que fizemos isso de forma manual para não colocar os segredos no código de configuração da infraestrutura. Existem algumas alternativas para gerenciar esse tipo de informação nas ferramentas mais populares de infraestrutura como código. No Puppet, você pode usar o **hiera-gpg** (https://github.com/crayfishx/hiera-gpg) ou o **puppet-decrypt** (https://github.com/maxlinc/puppet-decrypt) . O Chef possui o conceito de um *data bag* criptografado que pode armazenar segredos de forma confiável. Em Java, a biblioteca **Jasypt** (http://www.jasypt.org/) provê uma forma de colocar configurações criptografadas em arquivos `.properties` para que eles não sejam armazenados em texto puro.

Outro princípio interessante que vai ajudá-lo a encontrar problemas no

seu processo automatizado é tentar evitar logar em um servidor de produção para realizar qualquer tipo de operação. Bloquear o acesso SSH ao servidor remove um importante vetor de ataque, uma vez que ninguém (nem mesmo você) será capaz de executar comandos logando diretamente no servidor.

É claro que o acesso direto é apenas um dos vetores de ataque. A melhor forma de abordar segurança é sempre usar um esquema em camadas. Coloque proteção em cada nível do seu sistema: nos componentes de rede, na sua aplicação, nos pacotes de software que você escolher, no seu sistema operacional etc. Também é importante que você se mantenha informado sobre vulnerabilidades recentes e esteja preparado para atualizar as versões do seus componentes sempre que um problema de segurança for encontrado e consertado.

Segurança é um assunto complexo e cheio de detalhes, por isso é importante você procurar mais recursos especializados. Para aplicações web, um bom ponto de referência para os tipos de vulnerabilidades mais comuns e formas de protegê-los é o **OWASP** (https://www.owasp.org) . Todo ano, o OWASP publica uma lista dos dez maiores riscos de segurança aos quais a sua aplicação pode estar exposta. Eles mantêm até uma lista de ferramentas de teste que conseguem automatizar a verificação de alguns problemas de segurança que você pode incorporar na sua pipeline de entrega.

8.10 Conclusão

Um dos principais objetivos desse livro foi introduzir técnicas e conceitos de DevOps e Entrega Contínua na prática. Por esse motivo começamos com uma aplicação não trivial em produção logo no segundo capítulo. Aprendemos como monitorar e receber notificações quando algum problema é detectado em produção. Também vimos como tratar a infraestrutura como código, o que nos possibilitou recriar o ambiente de produção de forma rápida e confiável. Conseguimos, inclusive, migrar nosso ambiente para a nuvem sem muita dor de cabeça, reaproveitando o código que construímos ao longo dos capítulos. Aprendemos como modelar nosso processo de entrega e implementar um processo automatizado de *build* e *deploy* com diferentes tipos de teste automatizados que nos dão confiança de ir para produção mais fre-

quentemente. Por fim, com uma pipeline de entrega, implementamos um processo de *deploy* contínuo em que qualquer *commit* no código da aplicação ou de infraestrutura pode ir para produção quando passa por todos os testes.

Isso serve como uma base sólida que você pode aplicar imediatamente quando voltar ao trabalho amanhã. Como existem muito mais coisas para aprender e a comunidade DevOps continua evoluindo e criando novas ferramentas, não daria para abordar tudo em um livro só. Por isso, neste capítulo discutimos brevemente outros tópicos importantes que você vai se deparar quando embarcar na jornada de entrega contínua.

Nossa comunidade está crescendo e o interesse no assunto aumenta a cada dia. Não se assuste se as ferramentas que apresentamos aqui evoluam ou novas práticas apareçam, pois é assim que criamos conhecimento. O importante é que você saiba onde e como procurar por recursos e que você compartilhe seus aprendizados com a comunidade: escreva *posts* no seu blog, apresente um estudo de caso em uma conferência ou em seu grupo de usuários, compartilhe suas experiências boas e ruins, ajude a melhorar nossas ferramentas contribuindo com código, corrigindo defeitos ou melhorando sua documentação, enviando perguntas e sugestões no grupo de discussão do livro e, além de tudo, mantenha-se conectado!

Referências Bibliográficas

[1] John Allspaw. *The Art of Capacity Planning: Scalling Web Resources*. O'Reilly Media, 2008.

[2] David J. Anderson. *Kanban: Mudança Evolucionária de Sucesso para seu Negócio de Tecnologia*. Blue Hole Press, 2011.

[3] Mauricio Aniche. *Test-Driven Development: Teste e Design no Mundo Real*. Casa do Código, 2012.

[4] Kent Beck. *Programação Extrema (XP) Explicada: Acolha as Mudanças*. Bookman, 2004.

[5] James Birchler. Imvu's approach to integrating quality assurance with continuous deployment. 2010.

[6] Jez Humble e David Farley. *Entrega Contínua: Como Entregar Software de Forma Rápida e Confiável*. Bookman, 2014.

[7] Lisa Crispin e Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2008.

[8] Timothy Grance e Peter M. Mell. The nist definition of cloud computing. Technical report, 2011.

[9] Scott Ambler e Pramod Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[10] Martin Fowler. *Refatoração: Aperfeiçoando o Projeto de Código Existente*. Addison-Wesley Professional, 1999.

[11] Paul Hammant. App config workflow using scm. 2013.

[12] Steve Matyas e Andrew Glover Paul Duvall. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[13] Daniel Romero. *Começando com o Linux: Comandos, Serviços e Administração*. Casa do Código, 2013.

[14] Rafael Sabbagh. *Scrum: Gestão Ágil para Projetos de Sucesso*. Casa do Código, 2013.

[15] Pramod Sadalage. *Recipes for Continuous Database Integration: Evolutionary Database Development*. Addison-Wesley Professional, 2007.

[16] Danilo Sato. Uso eficaz de métricas em métodos Ágeis de desenvolvimento de software. Master's thesis, 2007.

[17] ThoughtWorks. *ThoughtWorks Anthology: Essays on Software Technology and Innovation*. Pragmatic Bookshelf, 2008.

[18] Adam Wiggins. The twelve-factor app. 2012.

[19] John Willis. What devops means to me. 2010.